Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Pedro Castro Ferreira

Web XML IDE

August 2023

Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Pedro Castro Ferreira

Web XML IDE

Master dissertation
Master Degree in Informatics Engineering

Dissertation supervised by
José Carlos Ramalho
Pedro Rangel Henriques

August 2023

STATEMENT OF INTEGRITY

José Pedro Castro Ferreira

____

I would like to start by thanking Professor José Carlos Ramalho for the technical guidance and for constantly pushing me to present good results on this project. I would also like to thank Professor Pedro Rangel Henriques for the patience showed while helping me write this thesis, being available at all times and providing me with the opportunity to see this project tested in the real world.

The last five years were full of great memories that I will take with me for the rest of my life and I would like to thank the 46 people who made this journey with me from day 1. Thank you for putting up with me, for trusting me and for giving me the best moments of my life. What we built and accomplished will always be my proudest achievement of the last five years.

To my closest friends Nuno Costa and João Bigas, thank you for helping me escape when I needed to, focus when I wouldn't and believe when I couldn't. Every single one of my achievements is (at the very least) a little bit yours.

A very special thank you to Rosa and Manuel Silva for every conversation, for their support, their friendship and for giving me a second home.

I would like to thank my family for keeping me focused, for their undying support, for believing in me and for being my biggest inspiration. To my mother, Isabel, thank you for keeping me on the right track the whole time and for the love only a mother knows how to give. To my sister, Raquel, thank you for providing me with such a good example on overcoming challenges and working hard to achieve what I want. To my sister, Sara, thank you for showing me what hard-work looks like and making me laugh with all the good stories she has to tell. And last to my father, José, for being my biggest inspiration, for being my role model, for showing me what hard-work looks like, for making me who I am today, for never letting me slip away and for supporting me in everything I ever did.

A very special thank you to my girlfriend Filipa Antunes for her patience during this year, for her help with this thesis, for endlessly reading about XML while having no idea what it even means. For making me strive to be the best I can, for believing in me, for pushing me, for making me laugh and for being there for me unconditionally. I am so lucky to have found you.

I would like to finish this part of my thesis remembering my grandfather Gabriel, my grandmother Noémia, my godfather Jorge and my aunt Judite. They passed away in the last five years and unfortunately they're not here today to witness me reaching the finishing line.

This is in their memory.

ABSTRACT

This document is the report for a Master Thesis, included in the second year of the Master's Degree in Informatics Engineering at Universidade do Minho in Braga, Portugal.

The project consists on the design and development of a Web IDE for XML to support teaching annotation languages to students that deal with digital documents but are not related to computer programming. The developed tool must be easy to install and to use, considering it is made to help users that are not experienced with programming or annotation languages nor IDEs.

The goal of this project is to get an user-friendly, easy to learn, Web-based platform to assist in: creating well-formed XML documents, DTDs, XML Schemas and XML Stylesheets; converting DTDs to XML Schemas; validating XML documents according to given DTDs or Schemas; running XPath expressions on XML Documents; and automatically generating XML documents from a DTD or a Schema and some parameters.

The state of the art on XML IDEs was analyzed in order to prove the necessity for the proposed application. The different IDEs were analyzed regarding their features, UI/UX, ease-of-use, price and availability.

This document also contemplates the proposed approach for meeting the project's objectives, the plan for the development stage of this project, as well as the definition for the desired features, and the report for the development process itself, describing how the different features were implemented.

The developed application was compared to the analyzed state of the art IDEs and proved to meet the proposed objectives for the project.

The **WebXMLIDE** application is publicly available for free in the following link: `https://webxml.epl.di.uminho.pt/`.

Keywords: XML, XSL, DTD, XSD, XSLT, XPath, IDE, Web-platform, Schema, Documents

RESUMO

Este documento serve como relatório de uma Tese de Mestrado, incluída no segundo ano do Mestrado em Engenharia Informática na Universidade do Minho em Braga, Portugal.

Este projeto consiste no design e desenvolvimento de um Web IDE de XML para suportar o ensino de linguagens de anotação a estudantes que lidam com documentos digitais mas não são de áreas relacionadas com programação. A ferramenta proposta deve ser fácil de instalar e usar, considerando que é feita para ajudar utilizador que não são experientes em programação, linguagens de anotação ou IDEs.

O objetivo deste projeto é obter uma plataforma baseada na web, fácil de utilizar e aprender para ajudar: na criação de documentos XML, DTDs, XML Schemas e XML Stylesheets bem-formados; na conversão de DTDs para Schemas; na validação de documentos XML de acordo com DTDs ou Schemas; na testagem de expressões XPath em documentos XPath; e na geração automática de documentos XML a partir de DTDs ou Schemas e alguns parâmetros.

O estado da arte no que toca a IDEs de XML foi analisado a fim de comprovar a necessidade da aplicação proposta. Os diferentes IDEs foram analisados quanto às suas funcionalidades, UI/UX, facilidade de uso, preço e disponibilidade.

Este documento contempla ainda a abordagem proposta para o cumprimento dos objetivos do projeto, o plano para a fase de desenvolvimento deste projeto, bem como a definição das funcionalidades pretendidas, e o relatório do próprio processo de desenvolvimento, descrevendo como foram implementadas as diferentes funcionalidades.

A aplicação desenvolvida foi comparada com os IDEs do estado da arte analisados e provou corresponder aos objetivos definidos para o projeto.

A aplicação **WebXMLIDE** está publicamente disponível de forma gratuita no seguinte link: `https://webxml.epl.di.uminho.pt/`.

Palavras-Chave: XML, XSL, DTD, XSD, XSLT, XPath, IDE, Plataforma Web, Schema, Documentos

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

C

Content Delivery Network.

Cascading Style Sheets.

D

Document Type Definition.

H

HyperText Markup Language.

Hypertext Transfer Protocol.

I

Integrated Development Environment.

M

Mathematical Markup Language.

Mode-View-Controller.

O

Object-Relational Mapping.

S

Synchronized Multimedia Integration Language.

U

Use Case Definition.

User Interface.

X

Extensible Markup Language.

XML Path Language.

XML QUeries.

XML Schema Definition.

Extensible Stylesheet Language.

XSL Formatting Objects.

XSL Transformations.

1 INTRODUCTION

This first chapter will be about explaining the context and motivation of the project, as well as explaining the project's objectives, the document structure, the research hypothesis and the development approach to the proposed software.

When talking about the context and motivation to the project, it's explained why this project is viable and useful. Then, the main objectives of the project are defined (on later stages of the project, each of these objectives will be turned into a specific feature and/or requirement). For now, it's important to keep an unrestricted, not technically defined or restrained view of these goals.

The goal of this project is to prove the Research Hypothesis, defined in section 1.3.

The time planning and work-order for researching, developing the proposed software, writing of the Master's thesis report and validating results is described in section 1.4 as well as the proposed structure for this document.

1.1 CONTEXT

With the constant growing importance of technology, it came as no surprise that it has been, still is, and will go on being used to ease day-to-day tasks on all areas of life for workers and consumers. This growth in interest and use, led to the world becoming more and more digital, and out of all things that are being transformed into a new digital form, this thesis report will focus on documents.

Documents of any kind are something we are not strangers to. They can be public records, medical reports or any document that is used to share information with someone or to be stored. Before the technological advances mentioned earlier, we were used to seeing this type of documents in binding folders archived in a room with hundreds of other bindings or being sent over mail to other people or businesses, but now that nearly every service has taken a digital form, how are these documents being shared or archived? We can send e-mails or write documents in our computers containing information of any kind, sorted as we like, in any language we want, containing no rules, no standards and no proper validation. This would obviously be a problem for sharing important documents that need to follow certain rules to be valid. For example, when a doctor looks at a patient's exams results, information should be clear, every number should have a measure next to it so that there's no ambiguity and this way, mistakes are avoided. The same goes to any document that needs validation and structure for the information they carry, and this is where XML comes into play Ramalho and Henriques (2002).

XML (Extensible Markup Language) is a markup language that was created to ease information sharing between machines and people. It was designed to be suitable for web-usage and it offers solutions to the problems mentioned before by identifying documents with custom tags, validating documents and creating rules about the structure of a given type of document with a XML DTD Inc. (1998) or XML Schema Sperberg-McQueen and Thompson (2000).

As technology grows and the world is taking more and more advantages of this growth, many workers now face this paradigm-shift of having to go from working with a paper and a pen, to working with a mouse and keyboard, which in itself represents a difficulty for some of these workers who are not used to work with a computer in any way, specially considering they are expected to be able to generate well-formed, valid and structured XML files. The solution is very simple: prepare these workers to manage, create and validate XML documents. To do so, these new users (often beginners when it comes to using code editors) need an easy-to-understand, highly-available platform not only to learn but to work on.

With a quick search online, it's easy to notice a certain lack of good affordable options when it comes to XML IDEs. Some are expensive, some are only available in certain systems and others simply don't have some of the features a beginner needs to be able to learn and properly create and validate XML documents and Schemas. This lack of an affordable, easy-to-use and complete tool is the reason behind this thesis.

1.2 OBJECTIVES

This Master's Thesis project has the following objectives:

- Development of a Web XML IDE, easy to install, learn and use.
- Development of a platform that helps users on creating well-formed XML documents, as well as on validating documents according to a DTD or a Schema.
- Development of a platform that helps users create DTDs and Schemas, and allows for DTD to Schema conversion.
- Development of platform features that allow for statistical reports, automatic generation of XML documents from a DTD or a Schema and some parameters, and document visualization in 3 distinct ways: text, dev (text and annotations) and transform (transformed text).

1.3 RESEARCH HYPOTHESIS

This Master's work intends to prove it is possible to create a tool using web development that allows creating, editing and validating XML documents in an agile and natural way for non-professional users.

1.4 DEVELOPMENT APPROACH

The development of this master thesis is going to follow an iterative methodology throughout an academic year, composed of the following steps:

- Bibliographic study and research on the state of the art on XML IDEs, their features, pricing and availability.
- Research of cutting-edge technologies that are going to be used to complete each goal of this project. More specifically, the implementation of a web app, creating and validating documents, code analyzers and compilers.
- Analysis over XML documents, DTDs and Schemas. These will be used later for testing in later phases of the development.
- Development of the proposed tool and all its requirements.
- Results evaluation and discussion

1.5 DOCUMENT STRUCTURE

This document is structured in 8 chapters. This first chapter presents this Master work's context, motivation, objectives, research hypothesis and development approach.

Chapter two will present a quick explanation of the most important components of XML documents and the tools that can be used to validate, present and explore these documents.

Chapter three will focus on the state of the art.

Chapter four will explain the proposed approach to develop the project.

Chapter five will present the development process (technologies used, plans and implementation).

Chapter six will provide screenshots of the final product and the list of implemented features.

Chapter seven will showcase the testing phase for the final product and analyze results from user feedback.

Chapter eight will serve as conclusion to this document, as well as present the work plan for the project.

2

XML & COMPANY

This chapter will explain the main components of this Master's project: XML, DTDs, XML Schemas and XSL (XSLT, XPath, XSL-FO and XQuery). It explains what XML and XSL are and how to generate well-formed, and valid documents (as well as their components).

When thinking about any way of writing digital documents, it's easy to imagine a software where the document is written and its content is formatted using the tools provided by the software.

In the context of XML documents, the document content is written in the XML file, using annotations (which define interpretation and structure) and data (which is the actual content) - both covered and explained in section 2.1.1 of this document.

These documents don't have any formatting and that's where XSL comes in: used to format XML files and make them presentable on the web, just like CSS is used to style HTML files.

2.1 XML

The Extensible Markup Language (or XML) is a simple and flexible text format, designed to meet the challenges of publishing digital documents. It plays an important role in the exchange of data on the web (Quin, 2016).

It can be used to write documents containing a wide variety of data with all sorts of meaning, use and importance.

Example 2.1 represents a simple XML document containing information about a CD Catalog:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
  </cd>
  <cd>
    <title>Divide</title>
    <artist>Ed Sheeran</artist>
  </cd>
  <cd>
    <title>Let It Be</title>
    <artist>The Beatles</artist>
  </cd>
  <cd>
    <title>AM</title>
    <artist>Arctic Monkeys</artist>
  </cd>
  <cd>
    <title>Marshall Mathers LP</title>
    <artist>Eminem</artist>
  </cd>
  <cd>
    <title>Chinese Democracy</title>
    <artist>Guns'n'Roses</artist>
  </cd>
</catalog>
```

Example 2.1: cdcatalog.xml

The most important part for now is to consider that this document is **well-formed**. The next section will explain what this means in more detail.

2.1.1 *Well-formed XML Documents*

An XML document consists of two components: data and annotations. But what are these? This section explains these two components, as they are key to a well-formed XML document.

The data in an XML document is blocks of text. Other data types are external to the document but can be referenced using entities.

The annotations in an XML document describe its structure and provide an interpretation to its content. An annotation consists of: symbols marking the beginning of an element, symbols marking the end of an element, symbols to represent empty elements, references to entities, comments, special text sections limiters, document type declarations and processing instructions (Ramalho and Henriques, 2002).

A simple example of a well-formed XML document, using the traditionally used *"Hello World"*:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
    Hello World!
</document>
```

Example 2.2: Hello_World.xml

An annotation always start with the symbol '<' and ends with '>'. With this in mind, it's easy to notice the document in example 2.2 contains three annotations: "<?xml ... ?>, <doc> and </doc>.

The next sections will go through the different components of a well-formed XML file, describing and explaining them.

THE XML DECLARATION

For a document to be considered well-formed, it must always start with an XML Declaration. Usually, if there's anything before the declaration or the declaration is absent, any XML parser will throw an error when parsing the document.

An XML Declaration is a special annotation with the following syntax:

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

Example 2.3: The XML Declaration

Three attributes can be used in an XML Declaration:

VERSION This attribute's value defines the XML Version that's being used. At the time this document's being written, there are only two available options for this attribute: 1.0 and 1.1. This attribute must be present in all declarations.

STANDALONE This attribute is optional and can have two values: "yes" means that the document is self-contained, which means it doesn't have any external reference; "no" means the document contains references to external documents (for example: other documents).

ENCODING This attribute is also optional and defines the encoding used for the characters. The default value is "UTF-8".

COMMENTS

A comment can be used at any point in an XML document. It starts with the symbols '<!–' and ends with the symbols '–>'.

A short example of a comment in an XML file:

```
<?xml version="1.0" ?>
<!--This is a comment-->
<doc>....</doc>
```

Example 2.4: The XML Comment

There are some rules when using comments:

- Comments must always come after the XML Declaration;
- Comments can't be written inside an annotation;
- The character sequence '--' can't be used inside a comment (because it marks the end of the comment).

PROCESSING INSTRUCTIONS

Processing instructions are the only reminiscent of procedural annotation that remains in XML. A processing instruction is not part of a document's content, it is actually a direct instruction to the parser that something must be executed at that point when the document is being transformed.

A processing instruction start with '<?processor-id> and ends with '?>'. Notice how the XML Declaration shown previously is nothing but a processing instruction, with 'xml' being the processor id.

Next, an XML document containing some processing instructions is presented:

```
<?xml version="1.0" ?>
<catalog>
    <?html action="hr"?>
        <cd id="cd1" type="album">
            <title>AM</title>
            <artist>Arctic Monkeys</artist>
        </cd>
    <?html action="hr"?>
</catalog>
```

Example 2.5: XML file containing processing instructions

Example 2.5 contains two processing instructions, besides the previously explained XML Declaration:

<?HTML...?> Both processing instructions identify actions to be executed when the XML document is being transformed to HTML. In this specific case, the processing instructions state that when the parsing process of the XML document coincides with the process of generating the documents HTML version, two *hr* marks must be generated at the position where the processing instructions are written in the original document.

At this point in this document, everything that can appear at the start of an XML document (before the body of the document), has already been explained: processing instructions, comments and the XML declaration (the first two may appear anywhere in the document).

A well-formed XML document has many different components, which will be described in the following sections of the document.

ELEMENTS

An XML document is structured based on **elements** that define the logic blocks in which the global content can be partitioned. An element is composed by three parts: starting annotation, content and ending annotation.

The starting annotation start with '<' and ends with '>' while the ending annotation starts with '</' and ends with '>'. Both contain a name for the element they identify and locate in the text.

A quick example of an element in a small piece of text:

```
I've been to <place>New York</place>, <place>London</place> and <place>Paris</
   place>.
```

Example 2.6: XML Element example

In example 2.6, the words *New York*, *London* and *Paris* are noted as elements with type *place*.

An XML rule dictates the names specified in the starting and ending annotations must be the same. This is one of the key principles to a well-formed XML document.

An element must be contained within another element and the only exception applies to the root element. Some hierarchical structures can be recursive. This is, an element may contain a direct or indirect instance of itself. This possibility of the use of recursion will, most of the times, cause problems during the information processing stages but it is necessary to model certain types of information. This subject will be exemplified when presenting DTDs and Schemas.

There are certain rules that the name of an element must follow:

1. the first character must be a letter, an *underscore* or a colon sign;

2. the following characters can be letters, digits, *underscores*, colon signs, dots and hyphens;

3. there can't be white spaces in the name of an annotation.

XML is case sensitive. This means *place*, *Place* and *PLACE* refer different elements.

It is also important to point out that the content of an element can never contain the characters '<' and '>' since these mark the limits of the annotation itself. Instead, the user must use '&lt;' and '&rt;'. These will make any editor present them as the intended characters, but won't interpret them as annotation limiters.

It's also important to present the different types of content an element can have:

TEXT-BASED CONTENT When the content of the element is purely text - just like the elements *title* and *artist* in example 2.1.

MIXED CONTENT When the content of the element is composed by text and other elements.

EMPTY ELEMENT When the element has no content. This usually happens when the element is used by its position. For example to break lines or to draw an horizontal line - these are like the commonly known HTML elements *br* and *hr*. This can also be the case for elements that represent references.

ATTRIBUTES

An element may have one or more attributes which can be optional or mandatory. These attributes give a sense of description to the element they're associated to.

When wondering what separates attributes from elements, it's easy to compare it to the English language: elements are like nouns, attributes are adjectives. Let's see the following example:

```
This is a house       ---  <house>
This is a green house ---  <house color="green">
```

Example 2.7: XML Attributes

There's no limit to the number of attributes associated to an element.

Attributes always come in the starting annotation to an element since they describe the content that follows them. An attribute is defined by a name and a value. The name and value of an attribute must be separated by the sign "=" and the value must be enclosed in single or double quotes. The names of the attribute follows the same rules as the names of the elements. The value will always be the text enclosed in quotes.

There are some reserved attributes in XML, but considering this document is not an extensive XML guide, it will not explain further than the basic concept of an attribute.

WELL-FORMING RULES

After explaining the basic components of an XML document, it's time to enunciate a set of rules that must be followed while creating a well-formed XML Document:

AN XML DOCUMENT MUST ALWAYS HAVE AN XML DECLARATION AT THE START - It's not mandatory, but most parsers will not work as they should if the XML Declaration is absent.

A DOCUMENT MUST CONTAIN ONE OR MORE ELEMENTS - To be considered well-formed, an XML document must contain one or more elements. The first element, which limits the entire body of the document, is the root element and all other elements must be contained in it.

ALL ELEMENTS MUST HAVE STARTING AND ENDING ANNOTATIONS - Empty elements are the only exception to this rule whose annotations must be replaced by a single starting annotation that ends with '/>'.

ELEMENTS MUST BE CORRECTLY NESTED - Considering an element is opened by its starting annotation and closed by its ending annotation, this rule states that an element that is opened inside another element, must be closed inside that element and never out of it. Any sub-element must be opened and closed inside its root element.

THE ATTRIBUTE VALUES MUST APPEAR INSIDE DOUBLE QUOTES - If in any given situation, the value needs to contain double quotes, it is possible to use single quotes to limit the value.

2.1.2 *Valid XML Documents*

As seen in the previous chapter, creating a well-formed document is not a hard task. But a well-formed document on itself has no proper validation method and that is what this section will be explaining.

The set of rules used to validate a document define a class or a type of document and allows for further validation and processing of XML documents.

The definition of a document's type or class is called DTD (*Document Type Definition*) or XML Schema. The process of checking if a document is following the rules defined in its DTD or Schema is called **validation**. After going through this process and being validated, the document classifies as a **Valid XML Document**.

2.1.2.1 XML DTD

This section will briefly explain DTDs, how to create one and associate it with an XML Document. Since the purpose of this document is not to be an in-depth tutorial, it's highly recommended to check the references that explain this concepts in a more extensive way such as Ramalho and Henriques (2002) and Inc. (1998).

As seen previously, DTDs define rules for documents to follow in order to be considered valid. Let's take a quick example, showing how to define a rule for a document.

Let's say our document is a contact list. What this means is that our document will have contacts with their corresponding information. In this example, let's consider a contact is composed by a name, an email address and a phone number.

Let's define a contact in DTD format:

```
<!-- A contact is composed by a name, an email address and a phone number -->
<!ELEMENT CONTACT (NAME,EMAIL,PHONE)>
```

Example 2.8: DTD Element Definition

So, this states that in the XML Document, every **contact** element must contain 3 elements: *name*, *email* and *phone*. In this case, the element **contact** is classified as a **structured**, since it's content is formed by a combination of other elements. It could also be of other 4 types, as follows:

EMPTY the element must not have any content, only a positional sense

TEXT the content of the element must be only text (with a few restrictions)

MIXED the content of the element is composed by text and elements

FREE content has no restraint, it can be a mixture of text and a combination of any other elements defined in the DTD. This type of elements will only be used in the initial state of the development of an application.

Considering the previous example, the content is structured, which means it's composed by a combination of other elements. Since the elements are between parentheses and separated by a comma, that means the elements should appear in the order specified in the element definition. If the elements were separated by a vertical line ('|', that would mean that one of the elements would be the content, given they were presented as **alternatives**.

Let's now define the elements that make the content of the main element:

```
<!-- A name is purely text -->
<!ELEMENT NAME (#PCDATA) >
<!-- An email is purely text -->
<!ELEMENT EMAIL (#PCDATA)>
<!-- A phone number is purely text -->
<!ELEMENT PHONE (#PCDATA) >
```

Example 2.9: DTD Sub-Elements Definition

Example 2.9 shows how to define elements of textual content.

Let's now consider elements may have certain attributes. Taking the example of the contact list, let's say the **CONTACT** element **must have** an *IDENTIFIER* and that it **may have** a *TYPE*.

Example 2.10 shows how to define the rules for the attributes of an element:

```
<!-- Defining a list of attributes for element contact -->
<!-- The identifier attribute has type 'id' and is mandatory -->
<!-- The type attribute has an enumerated type with default value "pessoa" -->
<!ATTLIST CONTACT
            IDENTIFIER ID #REQUIRED
            TYPE (pessoa|empresa) "pessoa" >
```

Example 2.10: "DTD Attributes of an Element Definition"

As shown in example 2.10, to define a list of attributes for an element, the statement starts with **"<!ATTLIST"** and ends with **">"**. After the keyword **"ATTLIST"** comes the name of the element we're associating the attribute list with. Then, for each attribute, the definition follows the same syntax:

```
ati-id ati-type ati-class ati-default
```

where

ATI-ID is the attribute identifier

ATI-TYPE is the attribute type

ATI-CLASS states the class that the attribute belongs to

ATI-DEFAULT defines the default value for attribute i

The only available attribute types are:

- CDATA
- Enumerate
- ID
- IDREF

- IDREFS
- ENTITY
- ENTITES
- NOTATION
- NMTOKEN
- NMTOKENS

This report will not dive deeper into explaining each type since the given references make an amazing job at explaining them and the differences between them.

Attributes also have a class that is one of the following:

#IMPLIED - Attribute is optional when the element is used in the XML document.

#REQUIRED - Attribute is mandatory. The user must provide a value for this attribute in every instance of the element.

#FIXED - Attribute is constant and immutable. The value provided in front of the keyword **#FIXED** will have to be the same for every instance of the element.

Specifying a class when defining the attribute is optional. If not specified, the processing tools will assume the attribute as an instance of a **#IMPLIED** class.

The default values for attributes are strings contained in double quotes like shown in the example above.

Before going into associating a DTD with an XML Document, let's see a full DTD file built with the examples used thus far.

```
<!ELEMENT LIST (CONTACT)+>

<!ELEMENT CONTACT (NAME,EMAIL,PHONE)>

<!ELEMENT NAME (#PCDATA) >
<!ELEMENT EMAIL (#PCDATA)>
<!ELEMENT PHONE (#PCDATA) >

<!ATTLIST CONTACT
            IDENTIFIER ID #REQUIRED
            TYPE (pessoa|empresa) "pessoa" >
```

Example 2.11: Real DTD Example

Every statement in example 2.11 has been previously explained, except for the first one that contains a '+' sign. This sign is one of three occurrence operators:

? (0 OR 1 OCURRENCES) x? - the element defined by this expression must be composed by an element x, which is optional.

*** (0 OR MORE OCCURENCES)** x* - the element defined by this expression must composed by zero or more elements x.

+ (1 OR MORE OCCURENCES) x+ - the element defined by this expression must be composed by one or more elements x.

To associate a DTD to an XML Document, the DTD must be saved in a file with the **.dtd** extension. Let's assume the file is called *"example.dtd"*.

There are two ways to reference a DTD on a document. One is used when the file is locally stored:

```
<!DOCTYPE example SYSTEM "example.dtd">
```

Example 2.12: DTD Attributes of an Element Definition

The other one is used when the file is stored online:

```
<!DOCTYPE example SYSTEM "https://somewhere.online.com/example.dtd">
```

Example 2.13: DTD Attributes of an Element Definition

This declaration must come after the XML declaration and before the root element. It also specifies the root element as *'example'*.

2.1.2.2 XML Schema

XML Schema was developed to try to answer some problems felt when using a DTD. DTDs were a part of the original methodology on XML Document Validation but with the appearance of new standards and XML-associated languages like SMIL and MathML, it became obvious that DTD were not powerful or expressive enough, specially to specify restrictions on the content of XML Documents.

XML Schema has some advantages when compared to DTD:

IT IS WRITTEN IN XML - DTD has it's own syntax while XML Schemas are specified in the XML syntax. The user that knows XML won't have to learn a new syntax.

DATA TYPE SUPPORT - XML Schemas support the most common data types of any other programming language and allow the user to define new data types.

STRUCTURES AND CLASSES - Powerful class and type system that allows for the extension and re-use of structures in itself.

SUPPORT NAMESPACES - This means XML Schemas developed in different NameSpaces can be combined, making a case for the practical use of modularity.

MIXED CONTENT ELEMENTS - Provides great features for the treatment of elements with mixed content.

Let's present a first example of an XML Schema Definition or XSD, using the following XML Document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<car>
   <year>2002</year>
   <brand>Toyota</year>
   <model>Auris</model>
</car>
```

Example 2.14: XML Document Example

The following XML Schema turns the well-formed XML Document in example 2.14 in a **valid** XML Document:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="car">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="year" type="xs:int"/>
                <xs:element name="brand" type="xs:string"/>
                <xs:element name="model" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Example 2.15: XML Schema Example

The XML Schema Definition is also an XML Document with the XML Declaration at it's top and the root of an XSD is an element called *schema*. The NameSpace usually used for a Schema is declared in the opening annotation of the root element.

In the previous example, the element definitions for elements *year*, *brand* and *model* all have an attribute **type**. This is used to identify the allowed datatype for the content of that element. XML Schema data types can be of three types: Simple Data Types, Compound Data Types and Derived Simple Data Types.

SIMPLE DATA TYPES

Simple data types restrain the text that can appear as content of either an attribute or an element. The content for textual elements and attribute values is pure text. XML Schema's support for data type specification allows for restrictions on the text of these attributes and elements.

Here are some of the supported base data types that XML Schema allows users to use to restrain the content of elements or to derive other base data types from:

4 primitive types for character sequences (text)	
string	A finite sequence of characters
anyURI	A normalized web address
NOTATION	Link declaration for an external document in a different format (other than XML)
QName	A qualified name follows the rules of the name of the element or attribute
3 primitive types for numerical data	
decimal	A decimal number of arbitrary precision
float	A floating point number (according to standard IEEE 754-1985) of simple precision (32 bits)
double	A floating point number (according to standard IEEE 754-1985) of double precision (64 bits)
9 primitive types for time-related data and dates	
duration	A period of time
dateTime	A precise instant in time (in the Gregorian calendar)
date	A specific date in the Gregorian calendar
time	An instant in time (hours, minutes and seconds)
gYearMoth	A year and a month of the Gregorian calendar
gYear	A year from the Gregorian calendar
gMonthDay	A month and a day from the Gregorian calendar
gMonth	A month from the Gregorian calendar
gDay	A day from the Gregorian calendar

Table 1: Supported Base Data Types on XML Schema

COMPOUND DATA TYPES

The declaration of a compound data type is used to define an element of structured type. This allows the user to specify which are the child elements, how these are composed, how many there are of each one, their attributes, etc.

A good rule of thumb that can be used to know if an element should be declared as a simple or compound type is the following: *"If an element has child elements or attributes, then it's a compound type element."*

The example presented in the beginning of this section (Example 2.15) shows the declaration of compound type (the keyword **complexType** is used to declare these types).

```
<xs:element name="car">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="year" type="xs:int"/>
            <xs:element name="brand" type="xs:string"/>
            <xs:element name="model" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Example 2.16: Compound Type Declaration

This way, the content of element **car** is defined as a compound type which is, in this case, a sequence of three elements.

Another way of achieving the same result would be to define an abstract compound data type (in this case *carType*) and later defining the element **car** with type *carType*:

```
<complexType name="carType">
    <sequence>
        <element name="year" type="int"/>
        <element name="brand" type="string"/>
        <element name="model" type="string"/>
    </sequence>
</complexType>
<element name="car" type="carType"/>
```

Example 2.17: Abstract Compound Type Definition

This methodology, when used in the definition of an XML Schema, allows for some important features:

- It becomes possible to declare more elements of the defined abstract data type without repeating the block of code that defines it.

- At any time, it's possible to change the abstract type. These changes will be automatically propagated to all elements declared as having that type.

As seen before, a compound type, defined by **complexType** in XSD, specifies a structure that can be constituted only by a set of attributes or by a substructure composed of other elements.

For a compound type to be defined as a combination of other elements, one of the following three composition operators must be used:

SEQUENCE - Specifies content composed by various elements that must appear in the specified order.

CHOICE - Specifies content composed of one element, arbitrarily chosen from a set of alternative elements. Works as an alternative of the elements it contains.

ALL - All elements contained in this composition operator can appear once or no times at all and can appear in any order. This composition operator has some restrictions: this operator can only be used in the highest level of nesting and it's elements can only be of simple data types.

When writing an XML Schema Definition, the user may have to define things like: an element is optional, an element can appear exactly three times or an element can appear five or more times. To define things like these, the occurrence operators are used. XML Schema has a mechanism that allows the user to specify the number of occurrences of a given element and it's implemented with two attributes: **minOccurs** and **maxOccurs**. The first one specifies the minimum number of occurrences of the element and the second specifies the maximum number of occurrences of the element.

Here are some of the possible combinations of these operators, which can be defined to have the same behaviour as DTD's occurrence controls:

minOccurs	**maxOccurs**	Description	DTD
0	**1**	Element is optional	**?**
0	**"unbounded"**	Element can occur zero or more times	*
1	**"unbounded"**	Element can occur one or more times	+

Table 2: Occurrence Operators on XML Schema

DERIVED SIMPLE DATA TYPES

In an XML Schema specification it's possible to create new simple types from an existing primitive simple type. These are called **Derived Simple Data Types** and are specified using a restriction mechanism predefined in XML Schema.

Before showing the possible restrictions, let's see a quick example of a Derived Simple Data Type. Considering the previously shown example of an XML Schema Definition for the element **car**, there's some missing information about a car. For example, the month of the car. We can define the simple type **carMonth** to associate with elements of type **car**, based on the primitive simple data type **integer** with some restrictions, as follows:

```
<simpleType name="carMonth">
    <restriction base="integer">
        <minInclusive value="1"/>
        <maxInclusive value="12"/>
    </restriction>
</simpleType>
```

Example 2.18: "Derived Simple Data Type Definition"

This way, the integer that appears as content of the element with type **carMonth** must be within the interval [1,12].

Here are some of the restriction operators available on XML Schema:

MININCLUSIVE - Minimum value corresponding to the lower limit of a closed interval of numerical values

MINEXCLUSIVE - Minimum value corresponding to the lower limit of an open interval of numerical values

MAXINCLUSIVE - Maximum value corresponding to the higher limit of a closed interval of numerical values

MAXEXCLUSIVE - Maximum value corresponding to the higher limit of an open interval of numerical values

LENGTH - This operator allows for text length specification

PATTERN - This operator allows for the specification of a regular expression that the content of the object must follow

ENUMERATION - This operator allows for the specification of a list with the possible values for the content of the element

ATTRIBUTES

An XML Document may contain attributes that serve to describe the elements they're applied on. In XML Schema, the definition of an attribute is quite similar to the definition of an element. The keys differences are:

- It can't contain elements or child attributes
- It's always defined with simple data types (primitive or derived)
- It's impossible to define the order in which the attributes appear in the element they're applied to

The declaration of attributes shall be done inside the *complexType* element, after specifying the element's content:

```
<element name="...">
    <complexType>
        [Content Specification]
        [Definition of attributes]
    </complexType>
</element>
```

Example 2.19: "Attribute Declaration Position"

The general case for the definition of an attribute in XML Schema follows the following syntax:

```
<attribute
name="attributeName"
type="attributeType"
use="attributeClass"
default="attributeDefaultValue" />
```

Example 2.20: "Attribute Definition Syntax"

By omission, when the existence of an attribute is declared, it's presence in a document instance is optional. Despite the fact that attributes can only appear once in each element, sometimes it's necessary to specify if the attribute must appear or if it's optional.

The attribute **use** of XML Schema is used to specify the type of occurrence of an attribute. This attribute can take the following values:

REQUIRED - Attribute must appear in the document, mandatory.

OPTIONAL - Attribute is optional (default value for attribute **use**).

PROHIBITED - Attribute must not appear in the document, must be omitted.

In XML Schema, it's possible to provide **default** and **fixed** values for attributes (just like in DTD). The default value for an attribute can be defined by assigning the attribute **default** to the attribute definition in the XML Schema with the desired value inside double quotes. To define a fixed value, the process is the same, switching the **default** attribute with the **fixed** attribute. If an attribute's value is fixed and the XML Document specifies the same attribute with a different value than the specified in the Schema, the parser will return an error since the attribute specification states that the value for that attribute is the one defined in the Schema.

The only types allowed for attribute's content are the simple types, either primitive or derived. Considering the possibility for restrictions seen in the Simple Data Types, these restrictions can also be applied on the content of attributes by defining derived types with the intended restrictions.

EMPTY ELEMENTS

An empty element has no text or children elements. The only thing it may contain is attributes. Otherwise, an empty element is just a mark, which is rare but may appear in some applications.

The two different ways of declaring an empty element are shown in Example 2.21:

```
<!-- Starting with the definition of an empty element without attributes -->
<element name="mark">
    <complexType/>
</element>

<!-- Now the definition of an empty element with attributes -->
<element name="image">
    <complexType>
        <attribute name="path" type="string" use="required"/>
        <attribute name="format" type="TiFormat"/>
    </complexType>
</element>

<!-- For the example to be complete, the TiFormat Type must be defined -->
<simpleType name="TiFormat">
    <restriction base="string">
        <enumeration value="GIF"/>
        <enumeration value="BMP"/>
        <enumeration value="PNG"/>
    </restriction>
</simpleType>
```

Example 2.21: Empty Element Definitions in XSD

MIXED CONTENT ELEMENTS

A mixed content is a combination of children elements and text. The necessity for mixed content comes naturally when the author has the need to associate meaning to a chunk of text.

To indicate that a *complexType* has mixed content, the attribute **mixed** is added with value **true**. Everything else remains the same as defining a compound data type, allowing for the specification of order, alternative and freedom of occurrence to child elements as well as the use of occurrence operators.

GLOBAL ELEMENTS VS LOCAL ELEMENTS

It's important to distinguish the declarations of global elements and local elements:

DECLARATIONS OF GLOBAL ELEMENTS - Children of root element **schema**.

DECLARATIONS OF LOCAL ELEMENTS - Nested somewhere along the structure of a Schema and are not direct children of the **schema** root element.

Once a given element is declared globally, any other element of the compound type can user that declaration, creating a reference to it. This possibility becomes extremely useful when facing a document with structural blocks that repeat along it's body.

Any globally defined element can be used as the root of an XML Document. In other words, in XML Schema there's no explicit definition of the root element and every element declared as direct child of the **schema** root element can be the root of document instances of that Schema.

2.2 XSL

XML Documents alone, present only the information they contain (annotations and data) without any formatting. This lack of a visual representation of data makes it harder for the common user to understand the content of a document. That's the importance of formatting. Making sure a document is easy to read is key when we consider the importance of the documents a user creates.

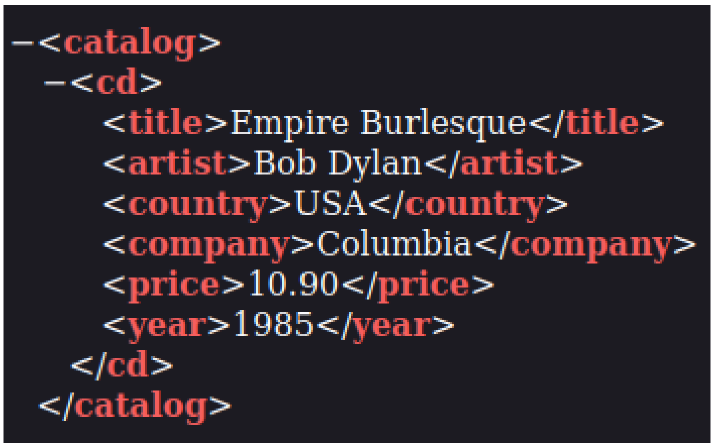
An XML Document without formatting presents information and structure but it may be harder to read if the user is not used to seeing documents structured in such way.

Consider example 2.1 of an XML document representing a CD Catalog, now containing more information about each of the CDs:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
   .
   .
</catalog>
```

Example 2.22: cdcatalog.xml

When opening this document in a browser, the information would be presented to the user as follows:



```
-<catalog>
  -<cd>
      <title>Empire Burlesque</title>
      <artist>Bob Dylan</artist>
      <country>USA</country>
      <company>Columbia</company>
      <price>10.90</price>
      <year>1985</year>
    </cd>
  </catalog>
```

Figure 1: Raw XML Document presentation

It's easy to see the data and annotations for the "cd" element but what if the catalog had more than one entry as it did in the previous example? It's easy to understand that this way of presenting information is not the cleanest way to present information to a user that's not used to work with XML. When writing an XML document, the writer wants the information to be easy to read and that's where formatting comes in.

Using XSL (and its components), it's possible to format an XML Document in such a way that allows the user to see the document as a web page. This will happen because the XML document will be **transformed** into another XML document or any type of document that is recognized by a browser like HTML. This transformation happens because of XLST, which will be covered in Subsection 2.2.1 (Quin, 2017).

Considering the previous XML document and it's presentation, let's now look at the corresponding XSL file, responsible for formatting the previously shown information:

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

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

<xsl:template match="/">
  <html>
  <body>
  <h2>My CD Collection</h2>
  <table border="1">
    <tr bgcolor="#9acd32">
      <th>Title</th>
      <th>Artist</th>
    </tr>
    <xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="artist"/></td>
    </tr>
    </xsl:for-each>
  </table>
  </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

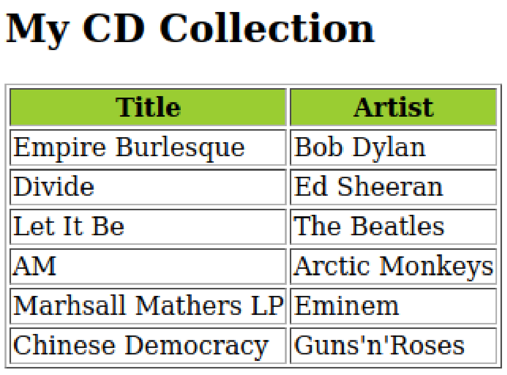
Example 2.23: cdcatalog.xsl

and adding the stylesheet reference to the original XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
   .
   .
</catalog>
```

Example 2.24: "cdcatalog_styled.xml"

This document now has a visual representation that makes information easier to read and comprehend (see Figure 2).



My CD Collection

Title	Artist
Empire Burlesque	Bob Dylan
Divide	Ed Sheeran
Let It Be	The Beatles
AM	Arctic Monkeys
Marhsall Mathers LP	Eminem
Chinese Democracy	Guns'n'Roses

Figure 2: XML Document Presentation after formatting

In the following sections, it is given a brief presentation of both XSLT and XQuery. There will also be a section about XPath, in which the language and its purposes are explained in a more detailed way than the other components of XSL.

2.2.1 *XSLT*

The example in Figure 2 shows an example of the result of applying a **style-sheet** to an XML Document. As previously mentioned, the definition of a visual representation of an XML Document is key to ensure readability and to ease communication between a publisher and a reader of a certain document.

To define these visual **transformations**, a **style-sheet** must be defined in the XSLT language, whose syntax is well-formed XML.

The term **style-sheet** reflects the fact that one of the key roles of XSLT is to add graphic information to an XML Document by converting the objects from the source document and transforming them into XSL formatting objects, or into another presentation-oriented format such as HTML.

There is a wide range of other transformations XSLT is used for (not exclusively for formatting and presentation applications). A transformation in XSLT describes a set of rules (called **template rules**) for transforming zero or more source trees into one or more result trees. The structure of these trees is described in the **Data Model**. A template rule associates a **pattern**, which matches nodes in the source XML Document with a **sequence constructor** (Kay, 2021).

As established previously, the purpose of this document is not to teach how to use these languages but to present them and explain their relevance to the project itself. To learn how to use XSLT, W3Schools offer's a good XSLT Tutorial[1].

2.2.2 *XPath & XQuery*

XPATH

XPath is an expression language that allows the processing of values conforming to the data model defined in the **XQuery and XPath Data Model 3.1**[2]. The name of this language derives from its most distinctive feature, the path expression, which provides a sense of hierarchy to the nodes in an XML Tree. As well as modelling the tree structure of XML, the data model also includes atomic values, function items and sequences (Jonathan Robie and Spiegel, 2021a).

Using XPath in XSL can have many objectives:

- Node selection for processing
- Specifying conditions allowing different processing modes for a single node
- Generating text to be included in the final tree

Many operations involved in these objectives involve choices based on the value of an element or an attribute or in a series of other factors.

XPath allows these choices to be defined in a syntax that is not XML-based. It is a syntax used to describe parts of an XML Document. With XPath, it is possible to reference every instance of an element, any attribute of an element, every element with a given content and many other variations. An XLST uses XPath expressions in two attributes, *match* and *select*, which are associated with various elements of XLST and give guidance to the document transformation.

Xpath was developed to be used as value of an attribute in an XML Document. The syntax is a mix of the expression language, normally used in other programming languages to describe arithmetic operations with values and variables with the language used to specify the path in a structure of directories like the one used in *Unix* and *Windows* Systems.

1 https://www.w3schools.com/xml/xsl_intro.asp, last accessed 26/11/2022
2 https://www.w3.org/TR/xpath-datamodel-31/, last accessed 26/11/2022

Additionally, XPath provides a set of text manipulation functions, *Namespaces* and other functionalities that are usually required when transforming a document (Ramalho and Henriques, 2002).

From the XPath's point of view, an XML Document is a Tree of Nodes. In the XPath Data Model, there are seven types of nodes:

ROOT NODE - Contains the whole document. In an XPath expression is always represented by '/'. Unlike other nodes, the root node does not have a parent node and has at least one child node, which represents the document. The root node can also contain comments and processing instructions (like the XML Declaration) that appear outside of the element that represents the document.

ELEMENT NODES - Every element in an XML Document is represented by a node. The children of an element node can be text nodes, element nodes, comment nodes and processing instruction nodes that appear inside the element in the original XML Document. The textual value of an element node is the concatenation of the text from this node and from all its children, by the order they appear in the document.

ATTRIBUTE NODES - An attribute node always has a parent node which is an element node. This type of nodes have some distinct functionalities that other node types don't have:

- Despite having an element node as the parent node, attribute nodes are not considered children of the same element node. If one wants to select the attribute nodes, it must be specifically instructed.
- In normal circumstances, the XPath processor creates an attribute node for every attribute that appear in the original document and for attributes with a default value defined in the DTD or XML Schema.

TEXT NODES - Text nodes are the simplest, containing only the text from the corresponding element. If the document contains entity references, these will be handled before the node is even created. A text node contains only pure text. A text node contains the maximum amount of text possible. This leads to these nodes not having sibling nodes of their type in the document tree.

COMMENT NODES - Also very simple, containing only text. The text of a comment node contains all the content of the original comment, except for the tags used to start and end a comment.

PROCESSING INSTRUCTION NODES - A processing instruction node has two parts: a name and a textual value. This value is everything that comes after the processing instruction name, except for the end tag '?>'.

NAMESPACE **NODES** - Nodes of this type are rarely used in XSL Stylesheets. These nodes exist mainly to benefit the processor, which has to differentiate elements that belong to different *Namespaces*. *Namespace* declarations in XML instances, even though they technically correspond to attributes, are handled in XPath as nodes of type *Namespace* (a declaration like: *xmlns:author="http://author.pt"*, would be stored in a node of type *Namespace*).

XPath works as a node selector in XSLT. Some of the XSLT commands use a selector to specify a node or a set of nodes to target for a transformation. The selector is an XPath expression.

A very important aspect of XPath is the concept of context. Context is the center piece of everything that is done with XPath. Comparing to a *Unix* or *Windows* file and directory system, context is like the current folder or directory. In this example, we can see the document tree like a tree of directories with files as the content for the children nodes. So, the context is a node of the document tree, from where an XPath expression is evaluated.

There are some different types of selectors. Let's start by presenting the **Simple Selectors** (see Table 3).

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

Table 3: XPath Simple Selectors

There are also relative, absolute, text, attribute and complex attributes. Table 4 presents a quick explanation on them.

Expression	Description
text()	Selects the textual content of a given element
/comment()	Selects comment nodes
/processing-instruction()	Selects the XML Declaration
*	Selects every element nodes in the current context
@*	Selects every attribute in the current context
node()	Selects every node of any type in the current context
//	When used in the middle of an XPath expression, indicates that between the double slash can occurr zero or more elements

Table 4: XPath Complex Selectors

So far, it's possible to select elements, attributes, text and comments with some simple XPath expressions, exploring, in most cases, the parent-child relation between elements. However, in many applications, it's necessary to select nodes in different perspectives like:

- Every ancestral node of the current node (parent, grand-parent,...)

- Every descendent node of the current node
- Every sibling node to the left (precedent) or to the right (following) of the current node

For this type of selection, XPath gives a mechanism called **navigation axes**. To use a navigation axis in an XPath expression, one must state the name of the axis, followd by a double colon ("::"), followed by the name of the target element. In total, XPath offers thirteen navigation axes described as follows:

CHILD Selects the children of the current node. In case of absence for the navigation axis, this navigation is used as a default value. The children node for the current node include element nodes, comment nodes, processing instruction nodes and text nodes. Attribute and *Namespace* nodes are not covered by the selection of this axis.

PARENT Selects the parent node of the current node, if it exists (if the current node is the root node, the result is an empty list of nodes). This axis can be shortened with the usage of the simple selector "..". The expression *parent::verse* and the expression *../verse* are equivalent.

SELF Selects the current node. This axis can also be shortened with the usage of the simple selector ".". The expression *self::** and the expression *.* are equivalent.

ATTRIBUTE Selects the attributes of the current node. If the current node is not an element node, the result will be an empty list. This axis can also be shortened with the usage of the selector "@". The expression */poem/attribute::type* and the expression *poem/@type* are equivalent.

ANCESTOR Selects every ancestral nodes all the way to the root node.

ANCESTOR-OR/-SELF The behaviour for this selector is the same as the previous, but adds the current node to the resulting list.

DESCENDANT Selects every descendant of the current node (children, grand-children, etc.). Only element nodes, comment nodes, processing instruction and text nodes are covered by this axis.

DESCENDANT-OR-SELF The behaviour for this selector is the same as the previous, but adds the current node to the resulting list.

PRECEDING-SIBLING Selects every precedent sibling of the current node. Selects every node with the same parent node as the current node and that appear before the current node in the original document. If the current node is an attribute or *namespace* node, the result will be an empty list.

FOLLOWING-SIBLING Selects every following sibling of the current node. Selects every node with the same parent node as the current node and that appear after the current node in the original document. If the current node is an attribute or *namespace* node, the result will be an empty list.

PRECEDING Selects every node that appear before the current node in the document, except for the ancestral nodes, attribute nodes and *namespace* nodes.

FOLLOWING Selects every node that appear after the current node in the document, except for the ancestral nodes, attribute nodes and *namespace* nodes.

NAMESPACE Selects every *namespace* nodes of the current node. In case the current node is not an element, the result will be an empty list.

One of the key concepts in XPath is the usage of **predicates**. A predicate can be defined as a filter that restricts the selected nodes by an XPath expression. Predicates are evaluated in runtime and return a boolean result. If, for a given node, the result of the predicate evaluation is true, the node is selected.

There are some things that can appear in a predicate:

NUMBERS - A predicate composed only by a number selects the nodes with that particular position. For example, the expression "*//chapter[1]/verse[2]*" selects the second verse of the first chapter. In reality, this type of predicates are an abbreviation of the positioning test of an element. Without these abbreviations, the previous expression would look like: "*//chapter[position()=1]/verse[position()=2]*".

ATTRIBUTES - A predicate composed by an attribute selection is true if the selected attribute exists in the current node. The expression: "*/poem/[@type]*" selects the poem, only if this element contains the *type* attributed.

FUNCTIONS - A predicate may contain the calling of XSLT functions. For example, the expression: "*verse[position() mod 2 = 0]*" selects every verse in an even position.

PREDICATE COMBINERS - XPath has some operators that allow for the combining of predicates: the boolean operators *"and"*,*"or"* and the union operator "|".

The last concept of XPath this report will present is **functions**.

XPath functions can be divided into four categories:

FUNCTIONS TO MANIPULATE LISTS OF NODES - Every function that works on the abstract document tree is contained in this category.

POSITION() - Result is a number corresponding to the position of the node in the document tree or, in case the function is being applied to the result of a sorting operation, the position in the sorted list.

LAST() - Gives as result a number corresponding to the total of existing nodes in the document tree at the same level as the current node. If used in a predicate, allows for the selection of the last node of that level, since the number of elements in a given level is the same as the position of its last node.

COUNT(EXP-PATH) - Gives as result the number of selected nodes by the XPath expression passed as argument.

ID(IDENTIFIER) - Gives as result the node that contains an attribute of type *ID* with value equal to *identifier*.

STRING MANIPULATING FUNCTIONS - This category contains functions that allow for text manipulation.

CONCAT(S1,S2,...) - Returns a string resulting from the concatenation of the various argument strings.

STARTS-WITH(S1,S2) - Returns true if *s1* starts with *s2*.

CONTAINS(S1,S2) - Returns true if *s1* contains *s2*.

SUBSTRING(STR,POS,LEN) - Returns a string extracted from *str*, starting on position *pos* with length *len*.

SUBSTRING-BEFORE(S1,S2) - Returns a substring of *str1* with every character that appears before the first occurrence of *str2*.

SUBSTRING-AFTER(S1,S2) - Returns a substring of *str1* with every character that appears after the first occurrence of *str2*.

STRING-LENGTH(STR) - Returns the number of characters in *str*.

NORMALIZE-SPACE(STR) - Returns *str* with normalized space: white spaces are removed at the beggining and end of the string. Also, all sequences of white spaces in *str* are replaced by a single white space.

TRANSLATE(S1,S2,S3) - Returns *s1* with the characters occurrence of *s2* by the character in the same position in *s3*.

BOOLEAN FUNCTIONS - This category contains every function that gives a boolean type result.

BOOLEAN(ARG) - Converts *arg*, which can be anything, in a boolean value.

NOT(BOOL-EXP) - Returns the inverse boolean value of the argument expression.

TRUE() - Returns boolean value *true*.

FALSE() - Returns boolean value *false*.

LAN(STR) - Returns true if the document's language is the same as the language passed as argument.

STRING(ARG) - Converts *arg*, which can be anything, to a string. When applied to an element with empty content, returns false, which gives the programmer a way to test empty elements.

NUMERICAL FUNCTIONS - This category contains functions that allow for arithmetic operations with element content.

NUMBER(ARG) - Converts *arg*, which can be of any type, to a number. If the argument is absent, tis function is applied to the current node.

SUM(XPATH-EXP) - Result is the sum of every element selected by *xpath-exp*, converted to a number. If one of the nodes has content that is not a number, the return value will be *NaN (Not a Number)*.

FLOOR(NUM) - Return the highest integer that is smaller or equal to the argument.

CEILING(NUM) - Returns the smallest integer that is not smaller than the argument.

ROUND(NUM) - Returns the closes integer to the argument.

XPath shows many possibilities to explore the document tree and process the content in a single node or in a group of selected nodes. It is an easier to learn, understand and use alternative to XQuery (which is based in XPath), so it will be the main focus of this project. A quick presentation of XQuery is made in the following section but in the development phase of this project, only XPath will be used as the query language when working with XML.

XQUERY

XQuery is designed to be a language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents. **XQuery** is designed to meet the requirement of having a non-XML query syntax.

This language is derived from an XML Query Language called **Quilt**, which borrowed features from several other languages like **XPath**, **XQL**, **XML-QL**, **SQL** and **OQL**. Jonathan Robie and Spiegel (2021b)

XQuery comes as an evolution from the previously presented **XPath** but considering the target audience for this project (which is inexperienced programmers, starting to learn XML and annotation languages), **XPath** emerges as the easiest alternative to learn and use to query XML Documents.

2.3 SUMMARY

To summarize, XML is annotation language that allows for creating digital documents. Languages like DTD and XSD can be used to define a validation schema for XML Documents. XSLT allows for styling XML Documents making them easier to visualize. And last, query languages such as XPath and XQuery can be used to explore and extract information from XML Documents.

3 STATE OF THE ART

This chapter will present a list of the most popular (according to (Help, 2022), and (Ghimire, 2022)) XML IDEs and Editors available at the moment this report is being written. For each IDE or Editor, let's analyze the features it offers, how relevant and easy-to-use these features are to a beginner-level XML user and how easy it is to install and use these tools.

After listing the XML IDEs and Editors, their features, pros and cons, a table will be presented, comparing all of the listed tools in terms of the previously mentioned characteristics among a few other considered relevant in the scope of this project.

3.1 OXYGEN

Developed in Java, **Oxygen XML**[1] is a cross-platform editor available on Windows, MacOS and Linux. Oxygen presents itself as the complete solution for XML Authoring, Development & Collaboration, offering nine different tools:

1. XML Editor
2. XML Developer
3. XML Author
4. XML Web Author
5. Content Fusion
6. Publishing Engine
7. PDF Chemistry
8. XML WebHelp
9. Feedback

1 `https://www.oxygenxml.com/xml_editor.html`, last accessed 29/11/2022

Of the above, this report will only focus on the first one when analyzing Oxygen's features, availability and ease-to-use since **XML Editor** is described as a union of **XML Author** and **XML Developer** by the Oxygen development team.

Let's take a look at the relevant features and technologies described in the XML Editor website.

STRUCTURED XML EDITING - A user-friendly interface is combined with a large number of intuitive XML editing features designed to help improve productivity and work quality.

INTELLIGENT CONTENT ASSISTANCE - Designed to save time and keystrokes when inserting repetitive or complex structures.

VALIDATION - Provides as-you-type validation support and context-sensitive editing capabilities to help the user make sure the documents are consistently well-formed and valid.

XML DATABASES SUPPORT - Allows for XQuery and XPath queries against a native XML database.

ALL XML STANDARDS SUPPORT - Oxygen XML Editor offers dedicated editors covering all XML Standards. The specialized views and operations of each editor offer support for editing all types of XML documents and other types of files like XML Schemas, XSLT, XQuery and many more.

XSLT & XQUERY DEBUGGING - Offers a powerful XSLT and XQuery debugger that provides full control over the debugging process. Two dedicated perspectives are available, one for XSLT and one for XQuery.

According to (Help, 2022), the biggest concern about this tool is the lack of a user-friendly UI, which in our case study, is one of the most important requirements for a beginner-friendly XML IDE.

One other key aspect about **Oxygen XML** is price. With prices ranging from 59$/year to 6576,5$/year (according to **Oxygen XML**'s own website), it's clear to see this tool is not affordable for everyone and, for an inexperienced user, this price is too high, considering most of the features **Oxygen XML** offers won't be relevant. The inexperienced user is looking for an XML IDE to learn how to write well-formed and valid documents and does not need the wide range of tools offered by **Oxygen**, making this tool's pricing, one of the biggest reasons why it isn't ideal for a beginner.

3.2 CODE BROWSER

According to **Code Browser**'s website[2], this tool is a folding text editor, available in Windows and Linux Systems as a stand-alone app. It is designed to hierarchically structure any kind of text file and source code. It is supposed to make navigation through source code faster, easier and more intuitive.

The basic idea of **Code Browser** is to only show the user what is relevant at each given moment. Let's say there's a big Java project with dozens of files, each containing a class and method definitions. **Code Browser** allows the user to see only the selected method from a selected class in a file. This makes code navigation a lot easier and reduces the amount of scrolling that is usually associated with big projects with massive source files.

Despite being very useful for massive projects, **Code Browser** does not offer the best features when it comes to a XML Editor or IDE. First of all, features like auto-complete, as-you-type syntax checking and syntax highlighting are missing. All these make all the difference for an inexperienced user that is trying to learn since it makes easier for the user to know what's doing wrong and to reduce the amount of keystrokes needed to create an XML Document.

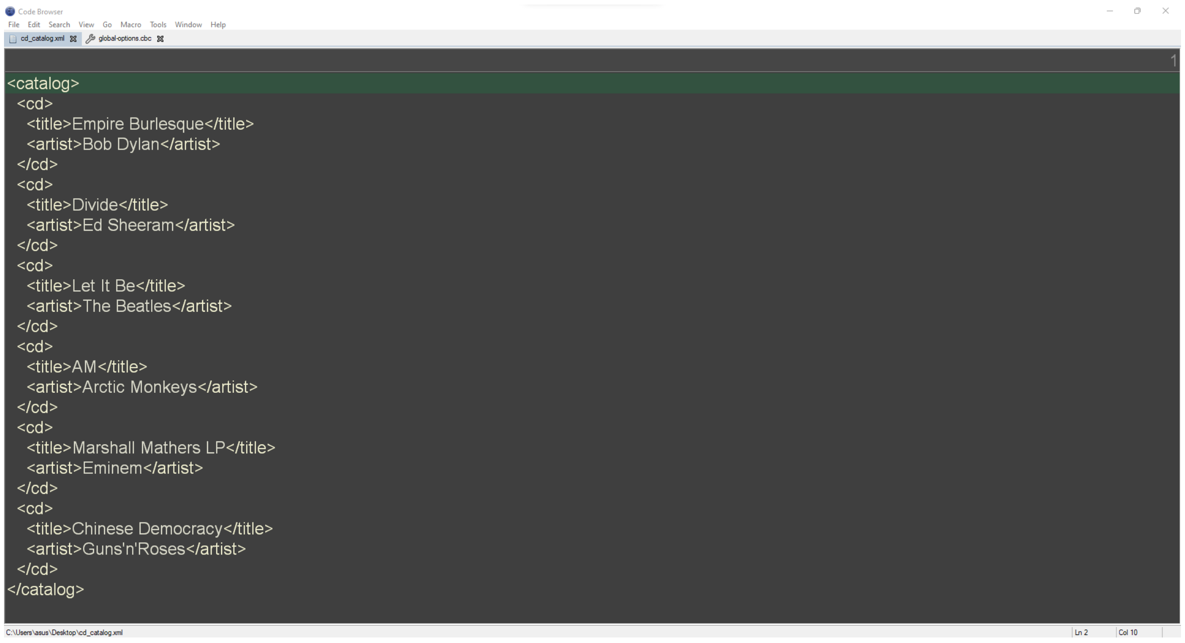
This tool is free and can be easily download and installed from the **Code Browser**'s website. The editor was tested using the CD Catalog (Code Example 2.1) file from the previous Sections.

Code Browser offers the chance to read and edit text and source files in a few different views:

- Simple
- Tree
- Smalltalk Style
- Zoomable

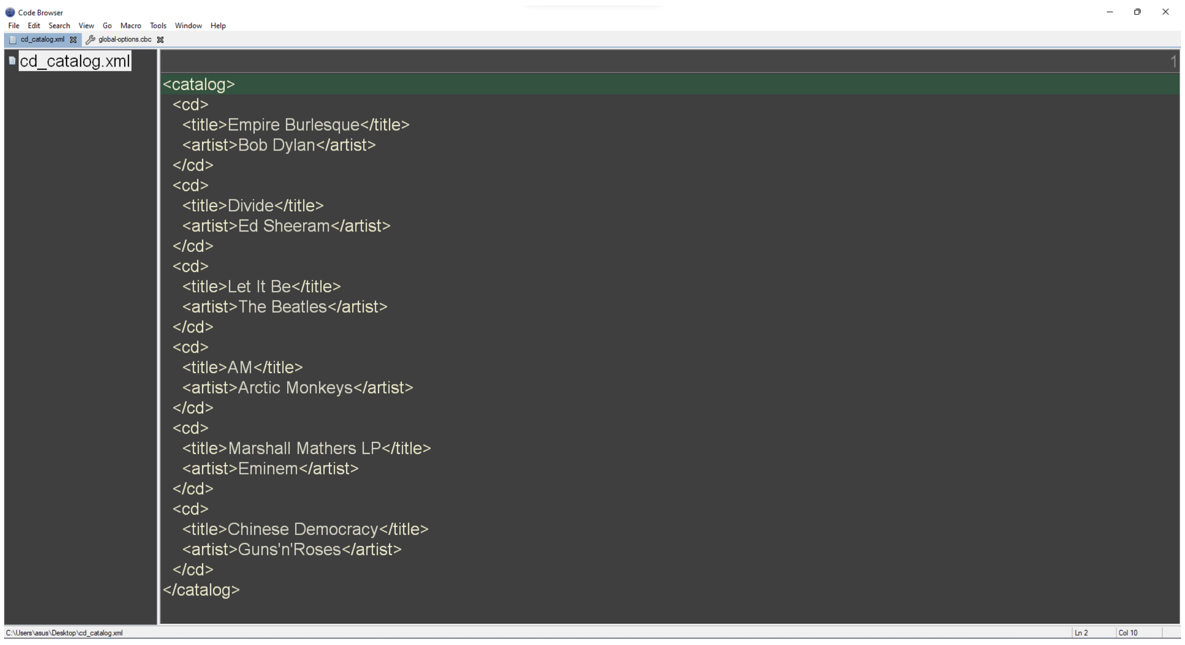
Let's take a look at how **Code Browser** presents *cd_catalog.xml* file in the 4 different views (see Figures 3, 4, 5, and 6).

2 `https://tibleiz.net/code-browser/index.html`, last accessed 02/12/2022


Code Browser
File Edit Search View Go Macro Tools Window Help
cd_catalog.xml
global-options.cbc
C:\Users\asus\Desktop\cd_catalog.xml
Ln 2
Col 10

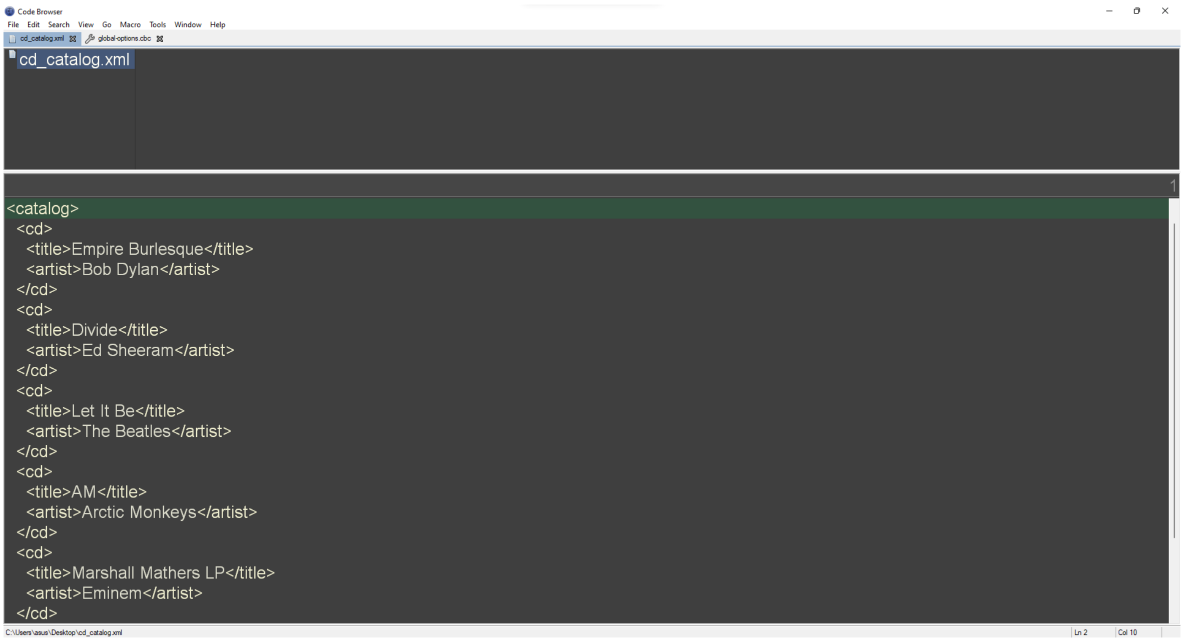
```
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
  </cd>
  <cd>
    <title>Divide</title>
    <artist>Ed Sheeram</artist>
  </cd>
  <cd>
    <title>Let It Be</title>
    <artist>The Beatles</artist>
  </cd>
  <cd>
    <title>AM</title>
    <artist>Arctic Monkeys</artist>
  </cd>
  <cd>
    <title>Marshall Mathers LP</title>
    <artist>Eminem</artist>
  </cd>
  <cd>
    <title>Chinese Democracy</title>
    <artist>Guns'n'Roses</artist>
  </cd>
</catalog>
```

Figure 3: Code Browser's simple view


Code Browser
File Edit Search View Go Macro Tools Window Help
cd_catalog.xml
global-options.cbc
cd_catalog.xml
C:\Users\asus\Desktop\cd_catalog.xml
Ln 2
Col 10

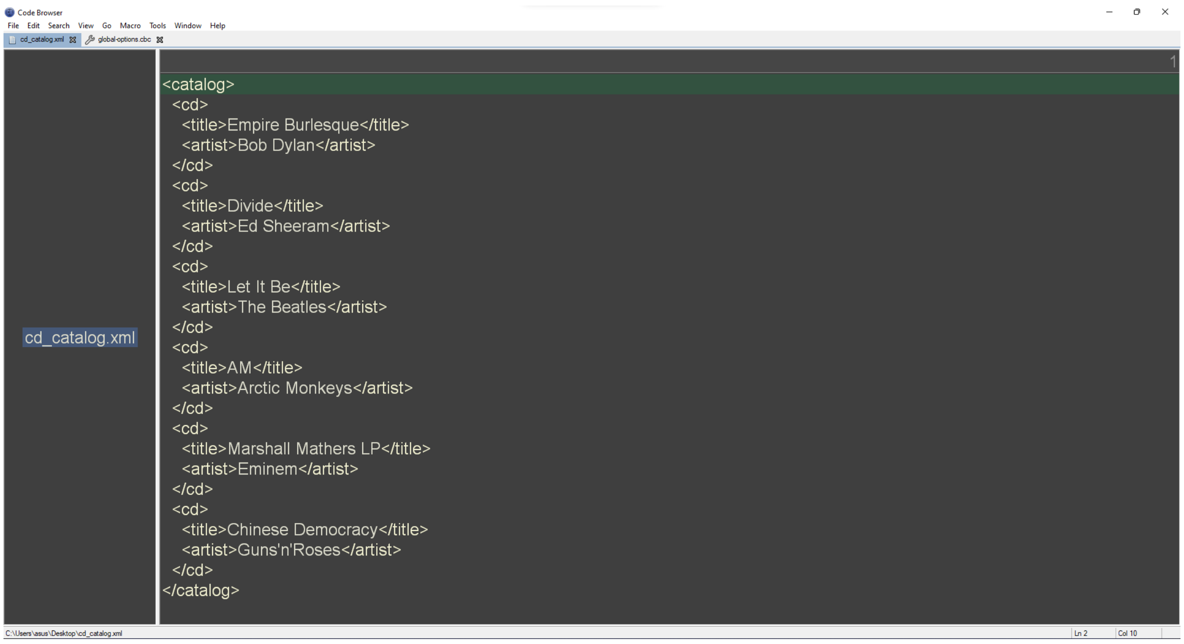
```
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
  </cd>
  <cd>
    <title>Divide</title>
    <artist>Ed Sheeram</artist>
  </cd>
  <cd>
    <title>Let It Be</title>
    <artist>The Beatles</artist>
  </cd>
  <cd>
    <title>AM</title>
    <artist>Arctic Monkeys</artist>
  </cd>
  <cd>
    <title>Marshall Mathers LP</title>
    <artist>Eminem</artist>
  </cd>
  <cd>
    <title>Chinese Democracy</title>
    <artist>Guns'n'Roses</artist>
  </cd>
</catalog>
```

Figure 4: Code Browser's tree view


Code Browser
File Edit Search View Go Macro Tools Window Help
cd_catalog.xml
global-options.cbc
cd_catalog.xml
C:\Users\asus\Desktop\cd_catalog.xml
Ln 2
Col 10

```
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
  </cd>
  <cd>
    <title>Divide</title>
    <artist>Ed Sheeram</artist>
  </cd>
  <cd>
    <title>Let It Be</title>
    <artist>The Beatles</artist>
  </cd>
  <cd>
    <title>AM</title>
    <artist>Arctic Monkeys</artist>
  </cd>
  <cd>
    <title>Marshall Mathers LP</title>
    <artist>Eminem</artist>
  </cd>
```

Figure 5: Code Browser's smalltalk view


Code Browser
File Edit Search View Go Macro Tools Window Help
cd_catalog.xml
global-options.cbc
cd_catalog.xml
C:\Users\asus\Desktop\cd_catalog.xml
Ln 2
Col 10

```
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
  </cd>
  <cd>
    <title>Divide</title>
    <artist>Ed Sheeram</artist>
  </cd>
  <cd>
    <title>Let It Be</title>
    <artist>The Beatles</artist>
  </cd>
  <cd>
    <title>AM</title>
    <artist>Arctic Monkeys</artist>
  </cd>
  <cd>
    <title>Marshall Mathers LP</title>
    <artist>Eminem</artist>
  </cd>
  <cd>
    <title>Chinese Democracy</title>
    <artist>Guns'n'Roses</artist>
  </cd>
</catalog>
```

Figure 6: Code Browser's zoomable view

The Figures above clearly show that **Code Browser**, despite being a really good option for big projects with many different source files, does not offer great support for single-document projects.

It is also worth mentioning the lack of a user-friendly UI, using the tool was very confusing and counter-intuitive.

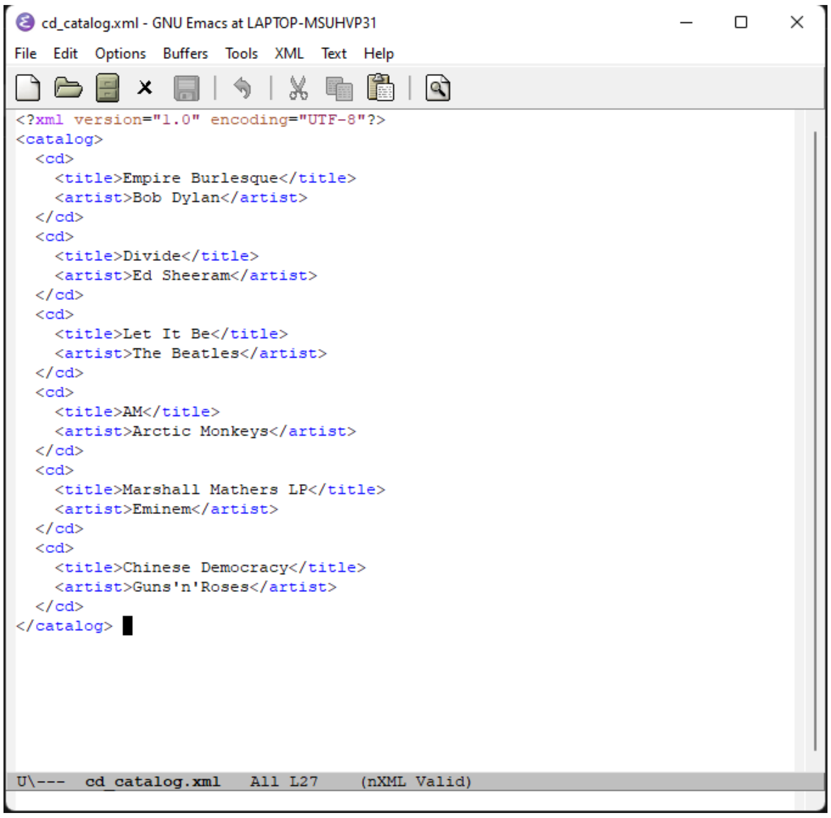
3.3 EMCAS FOR XML

Emacs[3] is a powerful text editor available as a stand-alone app for Linux, MacOS and Windows. This tool focuses efficiently the coding phase of XML files. It also has a specific XML tab with some options like *Validation, Schema Definition (from a file), etc.*.

Emacs is a free tool, available from the official website. Downloading and installing this software can be a bit tricky. An inexperienced user might not have the best time trying to download the installer, since the developers decided to give the user the ability to choose which version of **Emacs** to download, and for each version, there are a whole bunch of files that make the downloading process much more complicated than it needed to be.

Emacs offers a very intuitive UI (and simple, considering this is not an IDE but an Editor). It also offers very clean syntax highlighting and as-you-type error checking. This certainly helps beginners write well-formed documents. However, auto completion and code suggestion are features that would be key for this tool but are missing.

To test this editor, the same example file was used as in the previous Subsection (see Figure 7).


cd_catalog.xml - GNU Emacs at LAPTOP-MSUHVP31
File Edit Options Buffers Tools XML Text Help

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
  </cd>
  <cd>
    <title>Divide</title>
    <artist>Ed Sheeram</artist>
  </cd>
  <cd>
    <title>Let It Be</title>
    <artist>The Beatles</artist>
  </cd>
  <cd>
    <title>AM</title>
    <artist>Arctic Monkeys</artist>
  </cd>
  <cd>
    <title>Marshall Mathers LP</title>
    <artist>Eminem</artist>
  </cd>
  <cd>
    <title>Chinese Democracy</title>
    <artist>Guns'n'Roses</artist>
  </cd>
</catalog>
```

```
U\---  cd_catalog.xml   All L27    (nXML Valid)
```

Figure 7: Emacs for XML

3 https://www.gnu.org/savannah-checkouts/gnu/emacs/emacs.html, last accessed 02/12/2022

The downside for this tool is that is as simple as it gets. Only one file can be shown and edited at a time, making it hard to edit XML, XSD/DTD and XSL files. The fact we only get a development environment, lacking any kind of document visualization, makes this tool great for pure-XML writing but no so much for XSL integration, testing and debugging.

3.4 LIQUID XML STUDIO IDE

Classified as "**Best for beginners**" (Help, 2022), **Liquid XML Studio IDE**[4] is an application that focuses on providing the user with the most helpful features in an XML Development environment.

This software has an effective user interface, making it easier for users to work on it for long hours. The application also has features like split graphical and text view, making it easier to generate and supervise file transformations.

Liquid XML Studio IDE offers a wide range of features:

- XML editing and visualizing in tabular and tree grids for better code presentation
- Validates XML code against XML Schema
- Splits graphical and text views for better code management
- Syntax highlighting
- Multi-file editing
- Equipped with a real-time spell-checking feature
- Has an XML Sample Generator, allowing users to generate samples from the provided XML Schema
- Enhanced document formatting
- Auto-complete feature to complete the primary code section or provide suggestions

This tool is only available on Windows systems as a stand-alone app, available to download on **Liquid**'s website. According to (Kiarie, 2022), the most popular Operating System is still Windows accounting for around 72% of the desktop and laptop market. Even so, considering the desired tool should be available to everyone at anytime, this represents a restraint on the user's end, which is undesired.

Liquid XML Studio IDE is available for a free trial for 15 days after the initial activation. After this time the software will keep on operating as the Free Community Edition, which is

4 `https://www.liquid-technologies.com/xml-studio`, last accessed 02/12/2022

still a very powerful tool for beginners to use and learn from. The premium versions of the software cost between 139€ and 719€.

This tool is definitely one of the best (if not the best) seen so far in this report in the scope of this project. However, when looking at the user-interface and the amount of available options and features, an inexperienced user would be a little bit overwhelmed. At the same time it reduces dramatically the XML-learning-curve, **Liquid XML Studio IDE** requires that the user learns how to use a software that has so much to offer that is becomes too much for a beginner. A simplified, available as a web-app version of **Liquid XML Studio IDE** would be the ideal platform for a beginner to learn XML and the previously explained surrounding concepts.

3.5 STYLUS

Stylus[5] is an IDE for XML, considered by (Help, 2022) the best for support generator, with features like:

- Equipped with XML Parsing and validation architecture, making it easier for users to work along with XML Schema.
- Has a DTD validator that runs code against XML schema and ensures code efficiency.
- The self-indenting feature allows users to easily indent their code and enhance the presentation of their code.
- This tool is canonicalized and converts code into W3C canonical form, making it easier to understand and debug.
- In-built XML sample generator, making it easier for users to run through XML code and generate the most effective results.
- The folding code feature of this application allows users to code quickly.
- Equipped with a series of document wizards which allow users to extract data from various databases and files.
- The project window is integrated with source control systems and makes coding easier.
- Equipped with a Stylus studio and spell checker, allowing users to indent and check code quickly.
- This tool has an XML notepad to create a prototype of XML codes.

5 `http://www.stylusstudio.com/xml-download.html`, last accessed 06/08/2023

- This tool uses Document Object Modeler to work on the cross-platform interface and XML code.

One of the best features of **Stylus** is one that is not listed above. This powerful IDE gives the user the chance to view XML Documents in three different views:

TEXT VIEW - Used for code editing, folding and syntax highlighting

TREE VIEW - Useful when editing and working large XML files

GRID VIEW - Useful when doing calculations where you can view your XML file in a spreadsheet

This tool supports every component of XML and XSL covered in this document:

XML - XML Viewer, XML Parser and XML Validator

DTD - DTD Editor, DTD Validator, DTD Generator and DTD Standards

XML SCHEMA - XML Schema Editor, XML Schema Validator, XML Schema Generator and XML Schema Documentation

XSLT - XSLT Editor, XSLT Debugger, XSLT Designer, XSLT mapper and XSLT Preview

XQUERY - XQuery Editor, XQuery Debugger, XQuery Mapper and XQuery Performance

XPATH - XPath Evaluator, XPath Editor and XPath Generator

This tool presents itself as a serious contender for the best XML IDE in the market for both experienced and inexperienced users. The three different views and all the support given at the moment of code-writing makes this tool very beginner-friendly and pleasant to use.

The fact that this tool is only available in Windows Operating Systems is a concern, since it represents a restraint on the amount of users that can and/or will use it. Another concern found about this tool is the pricing. This software is only available to use for free for a fifteen-day trial and the price for a full-access license ranges from 99€ to 695€, making this very powerful software, a very unaffordable product, specially considering when the user's main goal is to use this tool for learning purposes.

3.6 KOMODO

Komodo[6] is a multi-language IDE by **ActiveState** available for Windows, Linux and MacOS. It features a very intuitive layout and many other interesting features:

6 https://www.activestate.com/products/komodo-ide/, last accessed 06/08/2023

- Most enhanced code intelligence, including all the core features like syntax highlighting, spell check, and auto-code complete.
- Equipped with a series of testing features, making it easier for users to work and check code efficiency simultaneously.
- Has various add-ons for users to experience multiple embedded features in it.
- Live preview feature, which lets you reflect your code on the output screen (ideal when working with XSLT).

Komodo allows users to customize the working environment as they require, making it easier to work and manage projects.

The main focus of this IDE is its code intelligence. This tool makes writing code a very easy task with great syntax highlighting, code refactoring and auto-completion (specific support for XML dialects added via DTDs or XML Schemas).

This tool offers a free solution, ideal for beginners, with the main features desired in a beginner-friendly solution for an XML IDE. There are also some paid licenses, but let's ignore them since there's a free version that is good enough to use continuously.

This tool is very complete, offers great coding support and is highly available, considering it is free and available on the three most popular operating systems for desktops/laptops (Kiarie, 2022).

Despite being one of the best solutions available, **Komodo** has the type of problems that are usually present in multi-language editors and IDEs: good support for a lot of languages and an intuitive UI that lead to compromising in offering a complete working environment for the supported languages. **Komodo** offers really good support when writing XML, XSL, DTD and XML Schema files, offers auto-completion on XML Documents according to dialects defined in DTDs and XML Schemas but it does not offer any form of validation on XML documents nor any form of document visualization other than the usual text view.

3.7 KATE

The **Kate**[7] editor features a debug window, a file explorer (makes working with multiple files easier) and plugins. The later are key when working on XML documents with **Kate**. The XML validation plugin will check for warnings and errors in XML files. **Kate** also has a plugin called *"XML Completion"* which checks whether the XML file follows a given DTD and verifies it (Ghimire, 2022).

This tools has a wide range of language compatibility and support, along with the possibility of using multiple tabs to see and edit different files at once. These features

7 `https://kate-editor.org/`, last accessed 06/08/2023

make **Kate** ideal when working with multiple files in the same project, even when working in multiple languages at once. Considering a beginner-level XML project with an XSLT stylesheet and an XML Schema Definition file, this software offers great support to work with all at once.

Kate offers syntax highlighting for over 300 languages and as-you-type spellchecking.

With a very intuitive and simple UI, **Kate** is available for Windows, MacOS and Linux systems for free.

As a multi-language editor, **Kate** lacks the same XML-specific features as **Komodo**. There are plugins to solve this lack of specific features but in a project where the target audience is one of inexperienced users, these features should be included in the default version of the tool, freeing the user from the responsibility to search and find the right plugin for a desired feature among the thousands of available plugins.

3.8 XMLGRID.NET

XMLGrid[8] is an online XML code editor. The tool allows the user to create an XML document in the platforms editor or to upload a local XML document and work on it in the web-app.

This tools presents XML Documents in one of two ways: a Text View - gives the user the ability to edit the content of elements and attributes -, and a Grid View - shows the document as a tree, folded in a hierarchical structure, where elements with text content and attributes are shown in a table, following the more conventional presentation form of a Grid View.

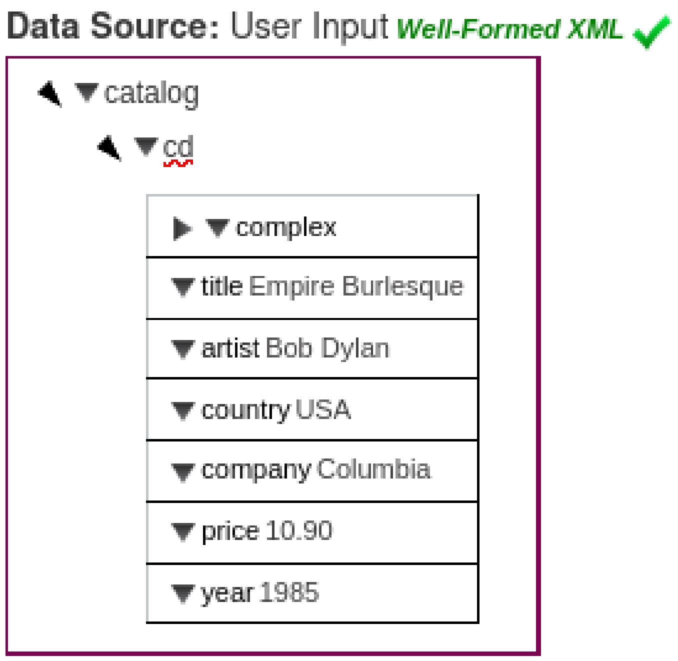
The XML Document in Example 3.1 was used to test this product.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <description>His name is <name>Slim Shady</name>, it's true!</description>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Culombia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
</catalog>
```

Example 3.1: "cd_catalog.xml"

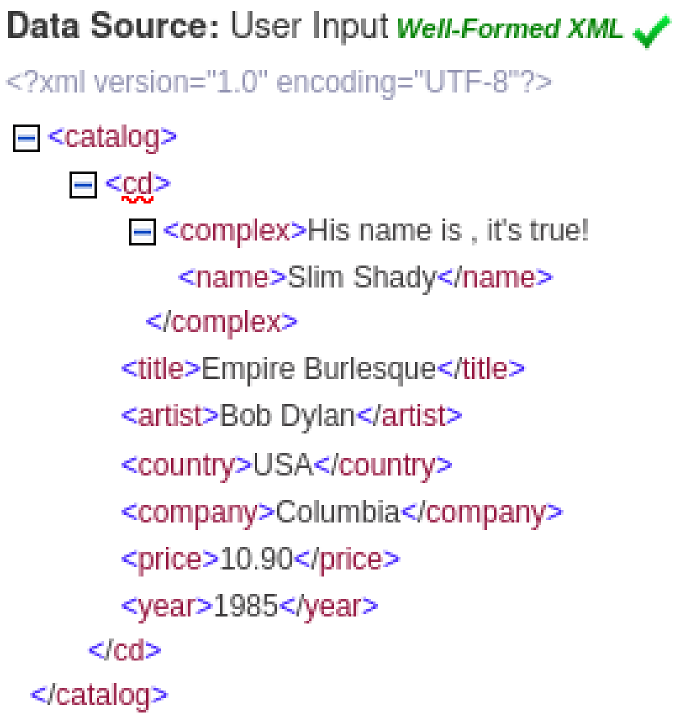
Uploading the XML file to the website or using the embedded editor generated the same output, which can be observed in Figures 8 (GridView), and 9 (TextView).

8 https://xmlgrid.net/, last accessed 06/08/2023


Data Source: User Input Well-Formed XML ✓

catalog
cd
complex
title Empire Burlesque
artist Bob Dylan
country USA
company Columbia
price 10.90
year 1985

Figure 8: Grid View Presentation


Data Source: User Input Well-Formed XML ✓

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
    <cd>
        <complex>His name is , it's true!
            <name>Slim Shady</name>
        </complex>
        <title>Empire Burlesque</title>
        <artist>Bob Dylan</artist>
        <country>USA</country>
        <company>Columbia</company>
        <price>10.90</price>
        <year>1985</year>
    </cd>
</catalog>
```

Figure 9: Text View Presentation

After spending some time exploring this tool, the features it offers, and working on the previously shown document, this tool feels very incomplete. UI feels very clustered, it is not very intuitive and it takes way too many mouse clicks to do pretty much anything. Also, features like automatic XSD generation from an XML Document, XML validation against XML Schema or DTDs, XPath Editor and others, are all shown in a very unorganized way at the bottom of the website's homepage.

Being the most available app seen so far considering it doesn't depend on the user's system (since it is an online tool), the user interface and user experience are clearly the biggest concerns about this tool. Editing a document is not as easy as it should, there is no syntax highlighting, auto-completion or code suggestions. Editing XSD or DTD is not an option, XSLT editing is not available and overall navigating the website is very difficult.

3.9 XMLSPY

XMLSpy[9] has various features like text and graphical editing split, making it easier for users to focus on code and make changes to it. This tool also provides enhanced intelligence guidance to work and debug the code.

This tool offers a really interesting list of features:

- XML Editing
- XML Schema Editor
- XSLT Editor, Debugger & Profiler
- XPath/XQuery builder & Evaluator
- XQuery Editor
- XSLT/XQuery back-mapping
- XPath/XQuery Debugger

This software offers different views for XML Documents, allowing the user to see text and graphical views at the same time.

XMLSpy features a complex UI, despite being easy to understand why. There are many features, many possible views and these are all visible options in the software UI. The ideal solution would have a simpler version of this tools interface, grouping features in menus and reducing the amount of visible options on the software window.

Being a really complete tool for XML development, **XMLSpy** has a free trial option that lasts for 30 days. To continue using the software after the trial, the user must purchase a license. Theses licenses are priced from 489€ to 839€, which is an unreasonable price for a beginner using this tool for learning purposes. This software is only available on Windows Systems.

9 `https://www.altova.com/xmlspy-xml-editor`, last accessed 06/08/2023

3.10 EXTENDSCLASS

ExtendsClass[10] is a free toolbox for developers. It offers three XML tools:

RANDOM XML GENERATOR - Given a set of defined functions, define the structure of an XML Document and the type of content each element would take (a name, a number, a regex, etc.). Then, the tools generates the desired amount of XML Documents with random values for the defined elements.

ONLINE XPATH TESTER AND EVALUATOR - Given an XML Document, write an XPath expression and this tool will check if the expression is valid or not. After checking for validity, the tool will run the expression on the XML Document and returns the result.

XSD VALIDATOR - Given an XML Document, generates an XSD for that document.

Offering only these features, it's easy to see that this tool has very specific use cases. Despite being really intuitive and offering really useful XML Development features, the lack of a real XML Editor, XSLT Editor and document viewers, among many others, make this product insufficient when used to learn XML.

This tool is free and available as a series of web-apps.

3.11 EDITIX

Editix[11] presents itself as a Powerful XML editor. This software offers a really good list of features that cover the entire XML Workflow:

- XSLT Editor and Debugger (V1 V2 V3)
- XPATH (V1 V2 V3)
- XQUERY Editor
- Powerful Grid Editor
- Syntax error and document XPath Location on-the-fly
- Validation of XML Documents against DTD/XML Schema
- Validation for W3C Schema
- Context sensitive content assistant based on XML Schema and DTD
- XSLT Transformation

10 https://extendsclass.com/, last accessed 06/08/2023
11 https://www.editix.com/, last accessed 06/08/2023

- Powerful XPATH Builder
- Multiple criteria research (attribute, element, namespace...)
- Project Management
- Fully Customizable (User Preference, Application Descriptor)
- Open XML Format Editor
- Refactoring (Elements, Attributes, Schema Type, Template Name, Variables and Parameters)
- Visual Schema Editor
- Schema Generator (DTD, W3C XML Schema) from XML Documents
- Multiple Search File (with XPath)
- XML Instances generated by W3C Schema or DTD
- XML Differencing
- Convert DTD and XML Schema

Editix has a bit of a clustered UI with many file managing operations available in the top of the software's UI with a large set of icons. This software would benefit from a cleaner presentation, specially considering the fact that the XML work menus are really well-organized.

Editix offers two types of licenses: a Professional edition with prices ranging between 99$ and 379$ and an Academic edition with prices ranging between 39$ and 699$. The only free version available is a 30-day free trial.

This tool is available on Windows, Linux and MacOS as a stand-alone app.

Despite being a bit clustered with the file operation icons, the UI on this tool is one of the best seen so far in this report. With separate drop-down menus for each XML or XSL component, it's easy to understand where-is-what. Featuring a file explorer, a tree view for the selected document and a text view to edit the document, this software has a really clean and easy-to-use UI that still allows for customization. The user can add panels for XSLT, XPath, XQuery, XML Schema and DTD editing, validation and debugging.

3.12 XMLMIND

XMLmind[12] is a tool that provides a lot of very useful features, either for publishing, coding or validating XML Documents.

12 https://www.xmlmind.com/xmleditor/, last accessed 06/08/2023

Here are some of the most interesting features **XMLmind** offers:

- Supports DTD, W3C XML Schema & XSLT
- XML to HTML-based formats and PDF conversion
- Strictly validating, schema-directed editing (ensures the document is always valid) - It CAN NOT open non-well formed documents
- Styled View, Tag View, Tree View and Source View (No Grid View)
- As-you-write spell checking

The complete list of features can be found in **XMLmind**'s website.

This software is available for free for personal and open-source use. Other licenses can be purchased with prices ranging from 130€ to 30000€. **XMLmind** is available as a stand-alone app for Windows, MacOS and Linux.

This tool features a very clustered and confusing UI and lacks support for very important components of the XML Family like XPath. Despite having a great number of features, most of them are focused on the publishing phase of XML Development. This tool provides many features but it is not designed for an inexperienced user.

3.13 CODE BEAUTIFY

Code Beautify[13] offers many different and useful tools for various purposes. It's a web-app with over one hundred and fifty tools like Unit and Format Converters, Web Viewers and Editors, Programming Editors, Cryptography tools, Validators, and many others.

For XML Development, **Code Beautify** offers the following features:

- XML Converter to JAVA, JSON, YAML, CSV, TSV, HTML and Excel
- XML-XSL Transform
- XML Viewer and Editor
- XPath Tester
- XML Validator (does not mention DTD or XSD)

It's easy to notice the lack of validation tools following DTD or XML Schema Definition, but nearly every other component of the XML Development Family is supported in this web-app. Also, the fact that there are so many features and tools in this app makes it hard to know where-is-what and by separating the relevant XML features instead of grouping them

13 `https://codebeautify.org/`, last accessed 08/12/2022

and treating each programming or annotation language as separate work environments, this tool gives the impression of a very unorganized work environment.

Being an online tool, it servers one of the main goals of the proposed tool of this project by being available to any user at any time and by being a free-to-use tool. To use any of the previously described tools, the user just has to visit the **Code Beautify** website and search for the intended tool.

3.14 ONLINE XML TOOLS

Online XML Tools[14] offers a series of XML utilities that are very useful in the context of XML Development:

- XML Prettifying and Minifying
- XML Validation
- XML Syntax Highlighting
- XML Converter to Image, JSON, CSV, YAML, TSV, Base64 and Plain Text
- XML Editor

All of the above are very intuitive to use, just like the app itself. However, there is a clear absence of DTD, XSD and XSL support.

This tools offers no auto-completion, no automatic XML generation, no DTD or XSD Validation, no XPath evaluator or tester and no XSLT editing, debugging or preview.

It is a free-to-use web-app, which makes this tool accessible to all users.

Despite being really intuitive, having a very clean UI, a very organized workflow and being highly available, this tool lacks support for too many important components of the XML Family, making it a very incomplete solution for XML Development, even when used for learning purposes.

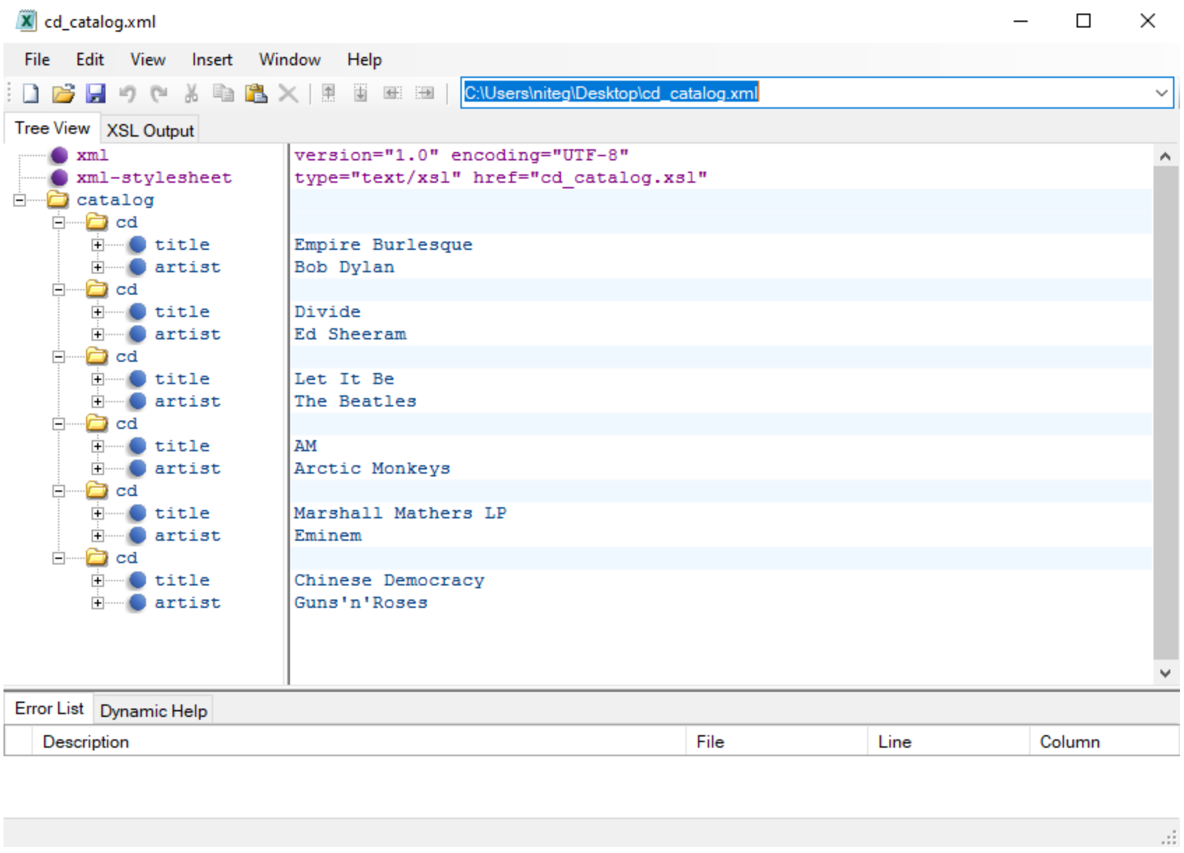
3.15 XMLNOTEPAD

XMLNotepad[15] is an open-source XML editor that offers a great UI that features in itself a text and a tree view of the document, an XSL preview tab and XSD validation.

Since this is an open-source and free-to-use app, let's test it with the CD Catalog example file used previously, as well as the associated stylesheet. To test this, we've stored both files on the same directory and opened the XML document with **XMLNotepad**.

14 `https://onlinexmltools.com/`, last accessed 06/08/2023
15 `https://microsoft.github.io/XmlNotepad/`, last accessed 06/08/2023

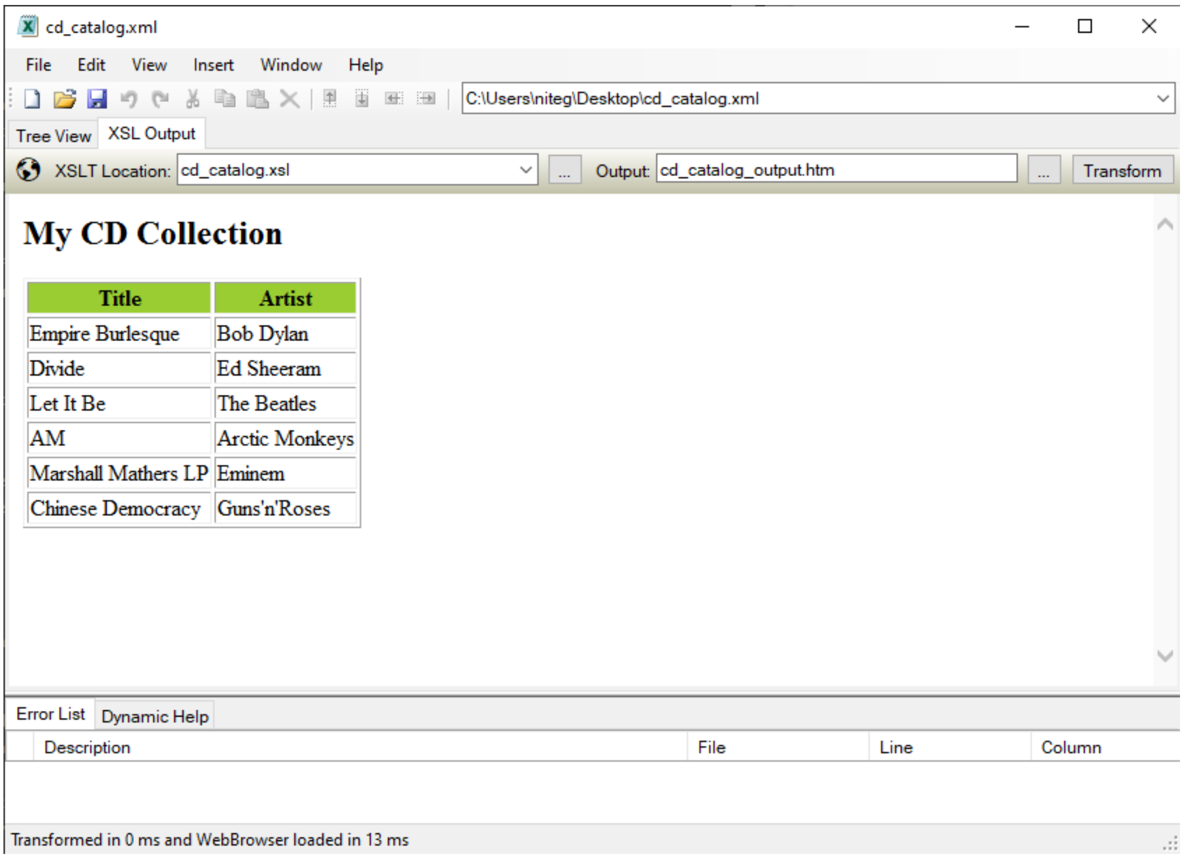
On the software's starting page, after loading a file, the user gets the document tree and text editor area to edit the content of elements and attributes (see Figure 10). To change the order of elements, the user can drag and drop the tree nodes around as it pleases. However, to change the order of the elements, to add an element or an attribute, the user can only do it with the drop down menu *"Insert"*, and it will insert either an element, an attribute, text, a comment, CDATA or a processing instruction which the user can edit. It is not possible to write code in this editor. Editing is a visual and graphical task instead of being a text one. The user can only edit content and can not edit the structural part of the document.



```
xml                   version="1.0" encoding="UTF-8"
xml-stylesheet        type="text/xsl" href="cd_catalog.xsl"
catalog
  cd
    title             Empire Burlesque
    artist            Bob Dylan
  cd
    title             Divide
    artist            Ed Sheeram
  cd
    title             Let It Be
    artist            The Beatles
  cd
    title             AM
    artist            Arctic Monkeys
  cd
    title             Marshall Mathers LP
    artist            Eminem
  cd
    title             Chinese Democracy
    artist            Guns'n'Roses
```

Figure 10: XMLNotepad's Main View

The page shown in Figure 11 allows the user to generate a preview of the XML Document, styled as it should by the referred stylesheet and generate an *.htm* file that can be previewed in any browser, with the resulting XSL Transformation of the XML Document. It is not very practical, considering the user can't edit the stylesheet on the software. Any changes have to be made using another tool or software which makes working on XSLT very inefficient.



My CD Collection

Title	Artist
Empire Burlesque	Bob Dylan
Divide	Ed Sheeram
Let It Be	The Beatles
AM	Arctic Monkeys
Marshall Mathers LP	Eminem
Chinese Democracy	Guns'n'Roses

Transformed in 0 ms and WebBrowser loaded in 13 ms

Figure 11: XMLNotepad's XSL Preview and Output

Despite having an intuitive UI and being free-to-use, in this software: code editing is non-existent, XSLT editing is non-existent, working with XML Schema is just as inefficient as working with XSLT, DTD is non-existent and the XPath evaluation tool is hidden within the *"Find"* option, usually used to find content by text or regex.

3.16 NOTEPAD++

NotePad++[16] is a free-to-use text and source code editor that supports syntax highlighting, code formatting, code folding and minor auto-completion for programming, scripting and markup languages (Ghimire, 2022).

Despite being really easy to use and having a really understandable UI, **NotePad++** does not feature code completion or syntax checking.

Some XML Tools can be added to **NotePad++** for editing XML Documents like the *"XML tools"* plugin, which can be found in the available plugins in the software's extensions list.

16 https://notepad-plus-plus.org/, last accessed 06/08/2023

This plugin provides XML, XSD and DTD validation by checking for format and syntax. It also supports XPath expression evaluation.

The native version (without plugins) alone, offers very little support to XML editing, besides syntax highlighting, code formatting and code folding. When analyzing how a beginner would use any of these tools, it is not convenient that the user should have to search for plugins and install them in order to user the given software or platform to learn the basics of XML. The user should be able to focus only on XML (and the XML Family) and not have to worry about searching for plugins or extensions. The *"XML tools"* plugin solves part of the lack of XML support but it is still not the best response for a complete XML Development Environment. Features like different document views, XSLT editing and preview, auto-completion for coding, etc. are all missing from this software.

3.17 VISUAL STUDIO CODE

Visual Studio Code[17] is arguably the most popular code editor today. It is free to use and available in Windows, Linux and MacOS. It offers a great UI and code editor for many different programming, scripting and markdown languages.

To use Visual Studio Code as an XML Editor, the user should install plugins to allow for syntax highlighting, a very good auto-completion for coding and some other tools like XPath evaluation and Text to XML Conversion (and vice-versa).

Visual Studio Code falls into the same problems other multi-language editors fall into. It does not offer great support on some features that are key to they XML Development Environment like XSLT preview, different document views, XSD or DTD Validation, etc.

3.18 RESEARCH SUMMARY

After going over the most popular XML Development Environments available and analyzing what makes each of them a good or a bad option for beginners to use as a learning platform, it's now time to define the system requirements, both functional and non-functional, for the tool proposed by this report, compare the previously analyzed tools and see how they perform on the defined requirements. Let's start by defining a list of requirements:

1. XML Code Editor
2. XML Syntax Highlighting, Auto-Completion and Code Folding (Editing features)
3. XML Document Views: Text, Dev (Text + Annotations) and Tree.

17 https://code.visualstudio.com/, last accessed 06/08/2023

4. XML Well-Formed Check
5. XML Validator according to XSD or DTD
6. Random XML Documents Generator according to XSD or DTD
7. XSD Code Editor
8. DTD Code Editor
9. DTD to XSD Converter
10. XSLT Editor
11. XSLT Preview
12. XPath Evaluator
13. High Availability (Web-App)
14. Price
15. Intuitive and simple UI/UX

In order to determine how good the previously analyzed tools are at satisfying this project's requirements, an evaluation system must be put into place to compare these tools. To have this evaluation system, the evaluation criteria must be defined. For each requirement, let's define levels of satisfaction and the sum of all these grades will be the final evaluation for each IDE, editor or toolbox.

The evaluation criteria that will be used is defined in Table 5.

Requirement	Criteria			
1. XML Code Editor	0 if absent		1 if present	
2. XML Syntax Highlighting, Auto-Completion and Code Folding	0 if none present	1 if 1 is present	2 if 2 are present	3 if all are present
3. XML Document Views: Text, Dev (Text + Annotations) and Tree	0 if none present	1 if 1 is present	2 if 2 are present	3 if all are present
4. XML Well-Formed Check	0 if absent		1 if present	
5. XML Validator according to XSD or DTD	0 if none present	1 if only one present		2 if both present

6. Random XML Documents Generator according to XSD or DTD	0 if none present	1 if only one present		2 if both present
7. XSD Code Editor	0 if absent		1 if present	
8. DTD Code Editor	0 if absent		1 if present	
9. DTD to XSD Converter	0 if absent		1 if present	
10. XSLT Code Editor	0 if absent		1 if present	
11. XSLT Preview	0 if absent		1 if present	
12. XPath Evaluator	0 if absent		1 if present	
13. High Availability (Web-App, Windows, Linux, MacOS)	0 if available only on 1 OS	1 if available only on 2 OS	2 if available on all 3 major OS	3 for WebApp
14. Price	0(Most Expensive) to 4(Free)			
15. Intuitive and simple UI/UX	0 to 10			

Table 5: Evaluation criteria for XML Editors, IDEs and Toolboxes

Considering the fact that some of these tools are really expensive and weren't tested, there's no way to know if some of the requirements are fulfilled or not by that tool. When this happens, instead of assigning a grade to the requirement, a null value will be assigned. In this case, the final grade will be an interval instead of a single integer, where the minimum will be the sum of all defined grades added to the minimum value of all null-defined grades and the maximum will be the sum of all defined grades added to the maximum value of all null defined grades.

When evaluating the pricing of each product (and considering most of the analyzed products have wide price ranges) the considered price will be the lowest available for that specific piece of software and every other requirement will be analyzed according to the cheapest version available. Also, for code editors that need plugins or extensions to support XML, this report will consider their basic installation (without any add-ons) since an inexperienced user should not have to worry about searching and installing plugins to work and learn XML.

Next, this report will show table with the evaluation of each product against the previously listed requirements and that product's final score when tested against the proposed features. These requirements are the ones intended for the tool proposed by this report. After evaluating every product and seeing which one satisfies this project's goals the most, this report will show a comparison between the "best" solution available and the intended solution.

Table 6 presents the evaluation of every IDE, Code Editor or Toolbox analyzed in this section against the intended features for the proposed tool.

Software	Requirements															Total
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Oxygen	1	3	-	1	2	0	0	0	0	1	1	1	2	2	7	[21-24]
Code Browser	1	0	3	0	0	0	1	1	0	1	0	0	1	4	3	15
Emacs	1	2	1	1	1	0	1	1	1	1	0	0	2	4	7	23
Liquid	1	3	3	1	1	1	1	1	0	1	1	1	0	4	5	24
Stylus	1	2	3	1	2	1	1	1	1	1	1	1	0	1	5	22
Komodo	1	3	1	0	0	0	1	1	0	1	1	0	2	4	8	23
Kate	1	1	1	0	0	0	1	1	0	1	0	0	2	4	10	22
XMLGrid	1	0	3	1	2	0	0	0	0	0	0	1	3	4	2	17
XMLSpy	1	2	3	1	1	0	1	0	0	1	1	1	0	0	5	17
ExtendsClass	0	0	0	0	1	0	0	0	0	0	0	1	3	4	7	16
Editix	1	2	2	1	2	1	1	1	1	1	1	1	2	3	8	28
XMLmind	1	1	3	1	2	0	1	1	0	1	0	0	2	4	4	21
Code Beautify	1	0	1	1	0	0	0	0	0	0	1	1	3	4	7	19
Online XML Tools	1	1	1	1	0	0	0	0	0	0	0	0	3	4	8	19
XMLNotepad	0	0	2	0	1	0	0	0	0	0	1	1	0	4	9	18
NotePad++	1	2	1	0	0	0	1	1	0	1	0	0	0	4	5	16
VSCode	1	1	1	0	0	0	1	1	0	1	0	0	2	4	10	22

Table 6: XML Editors and IDEs Evaluation

Analyzing the table above, **Editix** stands out as the best evaluated tool, followed by **Liquid** and **Oxygen**, and it is quite simple to understand why it does. Unlike any of the other analyzed tools, **Editix** features every functional requirement listed in this report, even though some of them are not present at their full extension. **Editix** offers XML, XSD, DTD and XSLT Code Editors, XPath Evaluation, Syntax Highlighting and Code Suggestions, Dev and Tree Views, Random XML Generator according to XSD, DTD to XSD Conversion, XSLT Preview and it is available on the three major Operating Systems at a relatively low price when compared to others in this report.

Figure 12 shows the results of comparing **Editix** with the proposed tool, which should feature every listed requirement.

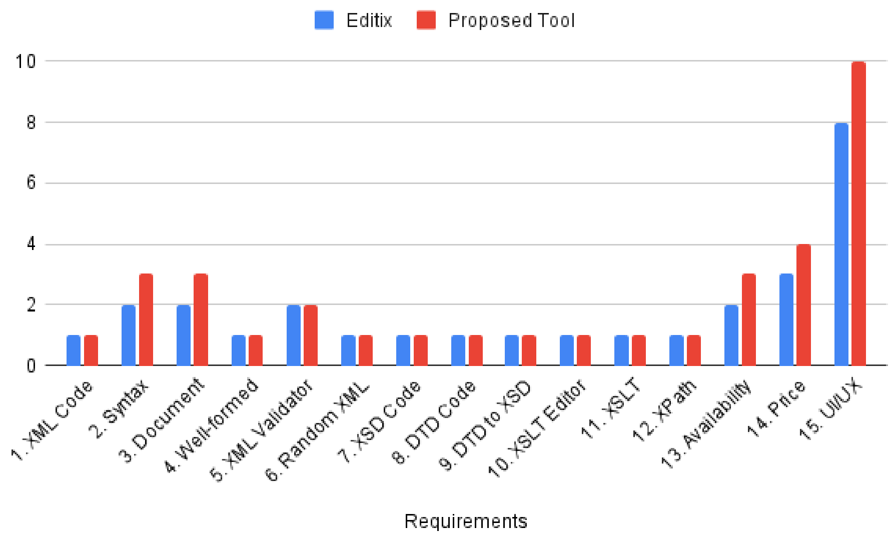


Figure 12: Editix vs Proposed Tool

Analyzing the difference between **Editix** and the proposed tool, the code editing features, document view modes, availability, price and UI/UX are what separates the already existing best option available from the tool this projects proposes.

CODE EDITING FEATURES - **Editix** offers good syntax highlighting and code suggestions. The proposed tool should offers these features and feature the possibility for code auto-completion. This is a feature that massively reduces the amount of necessary coder-side work and keystrokes. Exceptional when the user is inexperienced and specially considering the fact that XML Elements have starting and closing tags, this feature is key to ensure coding and learning is easy for the inexperienced XML user.

DOCUMENT VIEW MODES - Despite the fact that the Text View is somewhat present in the Dev view (Text + Annotations), **Editix** does not offer the possibility to view a document in it's text-form, which is something the proposed tool should be provide. The text view is very important when teaching XML to new users since it offers a chance of viewing an XML document's content, prior to any transformation, in a simpler way.

AVAILABILITY AND PRICE - **Editix** is available on the three most used operating systems. However, it still requires the user to download and install software, taking disk space and system resources to run. This project proposes a webApp to take the burden of running the software from the user's system, as well as eliminate all together the need for download and installation procedures.

UI/UX - Despite being a really organized tool when looking at the XML Work Environment Tools, **Editix** still features a confusing toolbar with many file managing, publishing and XML options that take a lot of space and attention away from the real work. If all these options were organized the same way as the toolbar menus for each component of the XML Family, **Editix**'s UI would be nearly perfect for a beginner-friendly XML Editor.

3.19 SUMMARY

To summarize the research part of this report, it's clear that some of the available options in the market are really good for XML users of any level. Despite the focus on **Editix** (which shown to be the best option available), other tools like **Oxygen**, **Emacs**, **Liquid**, **Stylus**, **Komodo**, **Kate** and **VSCode** are also really good options when choosing a platform to work on XML Development and Learning. However, there's a clear absence of good web-based options in this field of study. The analyzed web-based tools take 4 out of the 8 worst evaluations on the evaluation Table 6. This presents itself as an opportunity to develop a relevant, highly available, easy-to-use and free product that can be used by XML users of any level with a great set of features.

4 PROPOSED APPROACH

In this chapter, the system architecture will be presented, as well as the functionalities to which a user will have access.

The system architecture will be presented in the form of a Block Diagram.

The functionalities to which the users will have access, will be presented in the form of a Use Case Diagram. Every use case will later be explained in detail in Use Case Definitions.

4.1 SYSTEM ARCHITECTURE

As a web platform, the proposed architecture will be separated in three distinct components (named accordingly in Figure 13):

USER - The human user that interacts with the system by passing XML, DTD, XSD or XSLT code and XPath Expressions and receives the system's response after it is done processing information.

WEBAPP - Represents the platform's front-end. This layer will serve as an intermediary in the communication between the user and the functional system. This layer will provide the system with the user's input and will receive the system's output to present it to the user in the platform's UI. This is where the non functional requirements will present themselves: syntax highlighting, code folding, code auto-completion, XML Document Views, availability and UI/UX.

SYSTEM - Represents the platform's back-end. This layer will receive user input passed by the WebApp layer, process it and return the desired output by the user. This layer also has the ability to return any errors found during the input parsing and validating stages. This layer is where functional requirements are implemented: XML, DTD, XSD, XSLT Code Editor, XML Validation against DTD or XSD, Random XML Document Generation against DTD or XSD, DTD to XSD Conversion, XSLT Preview and XPath Expression Testing.

Figure 13 shows how the previously described layers interact with the user and how the system processes the information it gets from the WebApp.

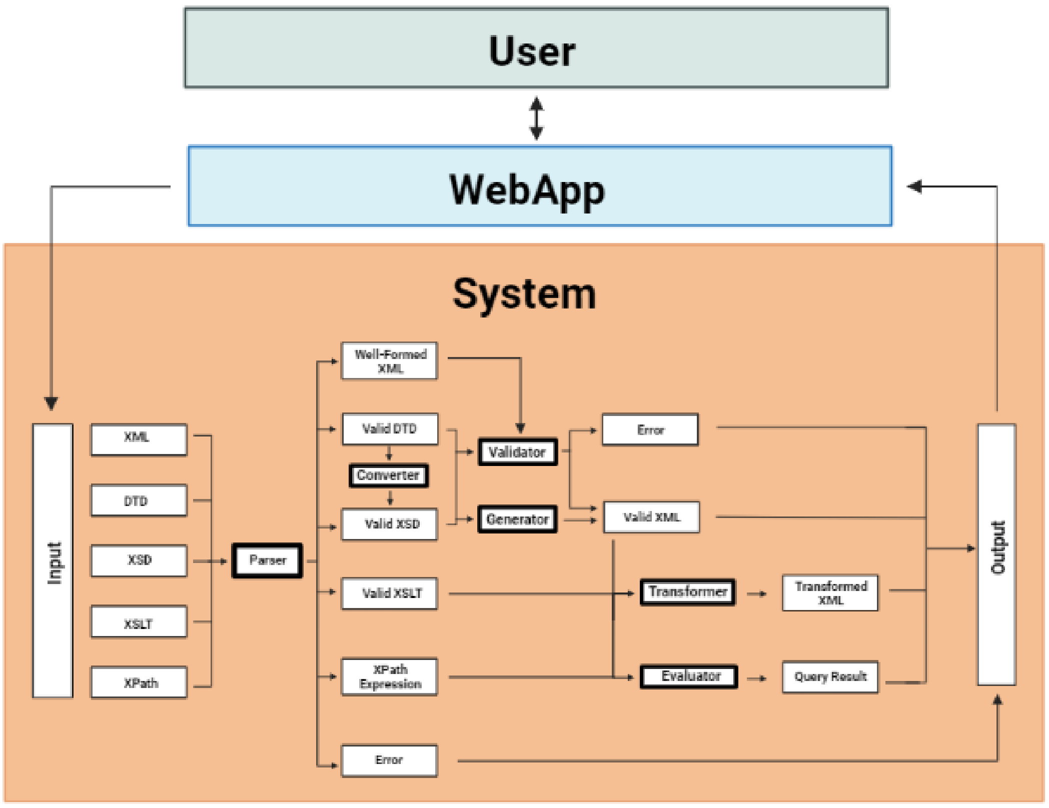


Figure 13: Proposed WebXMLIDE Architecture

The components inside the System layer represent three different aspects of information processing:

DATA - Input and Output; Input XML, DTD, XSD, XSLT and XPath; Well-Formed XML, Valid XML, DTD, XSD, XSLT, XPath Expressions, Transformed XML, Query Results and Errors; all of these represent data in some state. These forms of data are either the user's input, the result of processing other types of data or the final data collection to be passed back to the front-end.

PROCESSING UNITS - Parser, Validator, Converter, Generator, Transformer and Evaluator. All of these processing units represent a system functionality. These are responsible for processing the information they're passed and returning new forms of data, as well as any errors found during the processing stage.

COMMUNICATION - The arrows in the previous figure show how information flows within the back-end system.

4.2 USE CASES

In this section, some Use Cases are presented. These represent the functional requirements of the system:

1. Create Valid XML, DTD, XSD and XSLT Files
2. Validate XML Documents against DTD or XSD
3. Convert DTD to XSD
4. Preview XSLT Transformations
5. Teste XPath Expression
6. Generate XML Documents according to DTD or XSD

In this project, there's only one type of users and any user can do any of the previous Use Cases.

The Diagram in Figure 14 shows every Use Case, how they interact with each other and other system's functional components like the **Parser**, **Validator**, etc.

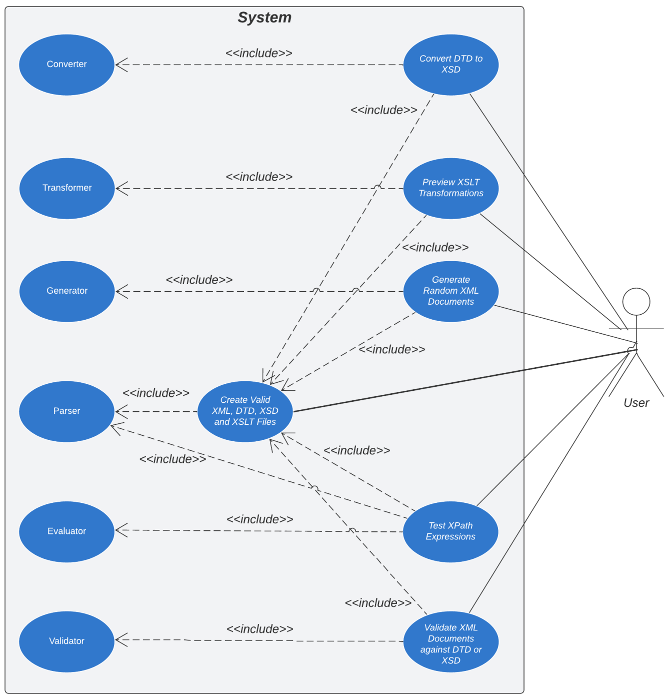

System
Converter
Transformer
Generator
Parser
Evaluator
Validator
Convert DTD to XSD
Preview XSLT Transformations
Generate Random XML Documents
Create Valid XML, DTD, XSD and XSLT Files
Test XPath Expressions
Validate XML Documents against DTD or XSD
<<include>>
User

Figure 14: Use Case Diagram

The diagram in Figure 14 shows the proposed Use Cases for this application and their relationships with each other. The following Subsections describe each Use Case in more detail.

4.2.1 *Create Valid XML, DTD, XSD and XSLT Files*

Table 7 shows the Use Case Definition for Creating Well-Formed XML Documents and Valid DTD, XSD and XSLT Files. This action can be performed by any user and requires no precondition. The result of performing this action should be a display of information stating that the XML Document is Well-Formed or that the DTD, XSD or XSLT File is Valid.

Use Case	1. Create Valid XML, DTD, XSD and XSLT Files	
Actor	User	
Precondition	None	
Post Condition	Files are Well-Formed or Valid	
Normal Scenario	Actor Input	System Response
	1. Writes XML, DTD, XSD or XSLT Code	
		2. Passes code to the Parser
		3. Notify user that documents are well-formed and/or valid
Alternative Scenario 1 [Parsing code returns errors] (Step 2)		2.1 Notify user that parsing returned errors
		2.2 List returned errors
	2.3 Edits code to fix errors	

Table 7: UCD - 1. Create Valid XML, DTD, XSD and XSLT Files

4.2.2 *Validate XML Documents against DTD or XSD*

Table 8 shows the Use Case Definition for Validating XML Documents against a DTD or XSD. This action can be performed by any user and requires only that the XML Document is Well-Formed and the DTD or XSD is valid. The result of performing this action should be a display of information stating that the XML Document is Valid against the specified DTD or XSD.

Use Case	2. Validate XML Documents against DTD or XSD	
Actor	User	
Precondition	XML document is well-formed and DTD or XSD is valid	
Post Condition	XML document is valid according to specified DTD and XSD	
Normal Scenario	Actor Input	System Response
	1. Checks for document validation	
		2. Checks if DTD or XSD is specified in XML Document
		3. Checks if XML Document follows DTD or XSD rules
		4. Informs user that the XML Document is Valid according to the specified DTD or XSD
Alternative Scenario 1 [DTD or XSD is not specified in XML Document] (Step 2)		2.1 Notify user that DTD or XSD is not specified in XML Document
		2.2 Suggests code to specify DTD or XSD in XML Document
	2.3 Edits XML File to specify DTD or XSD	
Alternative Scenario 2 [XML Document doesn't follow DTD or XSD rules] (Step 3)		3.1 Notify user that XML Document does not follow DTD or XSD rules
	3.2 Edits XML File to follow DTD or XSD rules	

Table 8: UCD - 2. Validate XML Documents against DTD or XSD

4.2.3 *Convert DTD to XSD*

Table 9 shows the Use Case Definition for Converting DTD to XSD. This action can be performed by any user and requires only that the specified DTD is valid. Performing this action should result in a valid XSD file that specifies the same rules as the specified DTD file.

Use Case	3. Convert DTD to XSD	
Actor	User	
Precondition	Input DTD is valid	
Post Condition	Results in a valid XSD	
Normal Scenario	Actor Input	System Response
	1. Requests DTD to XSD conversion	
		2. Converts DTD to XSD
		3. Presents the user the resulting XSD

Table 9: UCD - 3. Convert DTD to XSD

4.2.4 *Previewing XSLT Transformations*

Table 10 shows the Use Case Definition for Previewing XSLT Transformations. This action can be performed by any user and requires the XML Document to be well-formed and the XSLT Stylesheet to be valid. Performing this action should result in a preview of the transformed XML Document following the specified stylesheet.

Use Case	4. Preview XSLT Transformations	
Actor	User	
Precondition	XML Document is well-formed and XSLT Stylesheet is valid	
Post Condition	Transformed XML Document is presented	
Normal Scenario	Actor Input	System Response
	1. Request a preview of the transformed XML Document	
		2. Checks if XSLT is specified in XML Document
		3. Transforms XML Document into HTML-like output
		4. Presents the HTML-like generated output
Alternative Scenario 1 [XSLT is not specified in XML Document] (Step 2)		2.1 Notify user that XSLT is not specified in XML Document
		2.2 Suggests code to specify XSLT in XML Document
	2.3 Edits XML File to specify XSLT	

Table 10: UCD - 4. Previewing XSLT Transformations

4.2.5 *Test XPath Expressions*

Table 11 shows the Use Case Definition for Testing XPath Expressions. This action can be performed by any user and requires only that the XML Document is Well-Formed. Performing this action should result in a display of the query results to the user.

Use Case	5. Test XPath Expression	
Actor	User	
Precondition	XML Document is well-formed	
Post Condition	Presents result of applying XPath Query on the XML Document	
Normal Scenario	Actor Input	System Response
	1. Writes na XPath Expression	
		2. Checks if XPath Expression is valid
		3. Applies XPath Query on XML Document
		4. Presents Query Results
Alternative Scenario 1 [XPath Expression is invalid] (Step 2)		2.1 Notify user that XPath Expression is invalid
	2.2 Edits XPath Expression to be valid	

Table 11: UCD - 5. Test XPath Expression

4.2.6 *Generate XML Documents according to DTD or XSD*

Table 12 shows the Use Case Definition for Generating XML Documents from a DTD or XSD. This action can be performed by any user and requires only the specified DTD or XSD to be valid. Performing this action should result in the user being presented a random well-formed XML Document that is valid against the specified DTD or XSD.

Use Case	6. Generate XML Documents according to DTD or XSD	
Actor	User	
Precondition	DTD or XSD is valid	
Post Condition	XML Document is well-formed and valid against DTD or XSD	
Normal Scenario	Actor Input	System Response
	1. Requests for a random XML Document that is valid against given DTD or XSD	
		2. Generates Random XML Document valid against DTD or XSD
		3. Presents the user with the generated XML Document

Table 12: UCD - 6. Generate XML Documents according to DTD or XSD

4.3 DESIGN MOCKUPS

In order to make it ease understanding the tool's purpose and features, this section contains the mockups for the proposed web-app.

These mockups are not supposed to serve as a restraint on the final design for the web-app, but as a guide and a visual representation of the tool's features, intended interface simplicity and usability.

4.3.1 *XML Editor*

The first mockup (see Figure 15) shows the app's XML Editor. This editor will always be presented to the user, taking the left side of the screen. Here's how the XML Editor will be

presented:

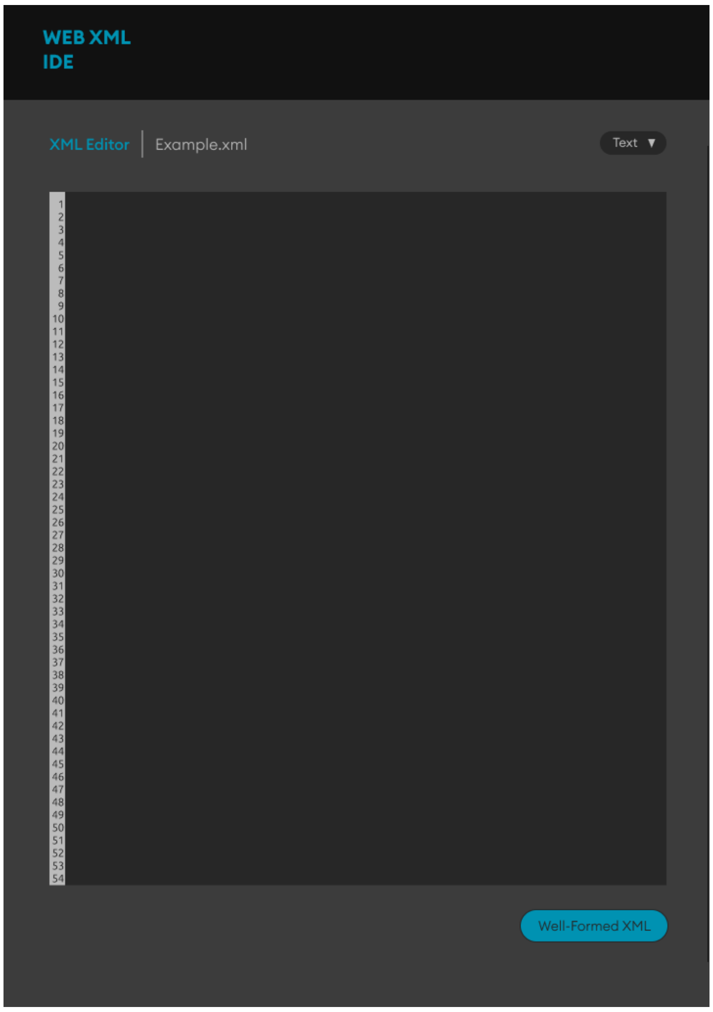


Figure 15: Mockup XML Editor

In the image of Figure 15, it's possible to identify a few elements:

FILE NAME - A form that allows the user to define the name for the XML file

VIEW MODE SELECTOR - A drop-down menu that allows the user to choose the view mode for the document: dev, text or tree.

XML CODE EDITOR - Standard code editor with syntax highlighting, allows for code folding and offers auto-completion.

WELL-FORMED XML CHECK - Visual indicator for the well-formation of the document

This editor and all of the previously described components will be presented to the user alongside every other component of the XML Family seen so far in this report.

In the next subsections, the mockups for all pages will be presented and described. Each page allows the user to work with XML and a single component of the XML Family.

4.3.2 *DTD Editor and Tools*

The mockup for the DTD Editor and Tools page of the proposed app is presented in this section. The left side of the screen is taken with the XML Editor described previously and the right side is a DTD workspace with some components that will be described later.

Figure 16 contains the mockup for the DTD Editor and Tools page.

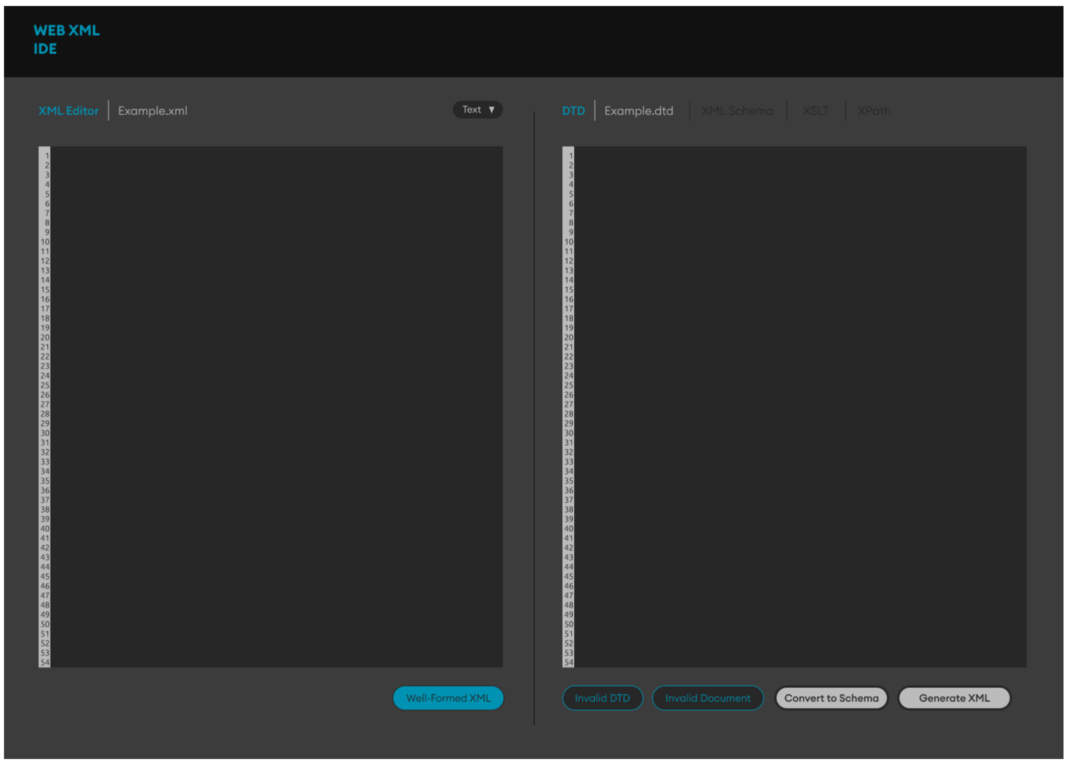


Figure 16: Mockup DTD Editor and Tools

Since the components of the XML side of the web-app have been described previously, let's focus on the components on the DTD workspace:

WORKSPACE SELECTOR - In this case, the DTD workspace is active but the selector allows the user to change workspaces in just one click.

FILE NAME - Allows the user to define a name for the DTD file.

DTD CODE EDITOR - Code Editor where the DTD is defined.

DTD VALIDATOR - Checks if the DTD defined in the code editor is valid.

DOCUMENT VALIDATOR - Checks if the XML Document on the XML Code Editor is valid against the defined DTD.

CONVERTER - Takes the specified DTD and converts it to a XML Schema Definition

GENERATOR - Takes the specified DTD and generates an XML Document that is valid against the same DTD, with random values for it's elements.

4.3.3 *XSD Editor and Tools*

The mockup for the XSD Editor and Tools page of the proposed app is presented in this section. The left side of the screen is taken with the XML Editor described previously and the right side is a XSD workspace with some components that will be described later.

Figure 17 contains the mockup for the XSD Editor and Tools page.

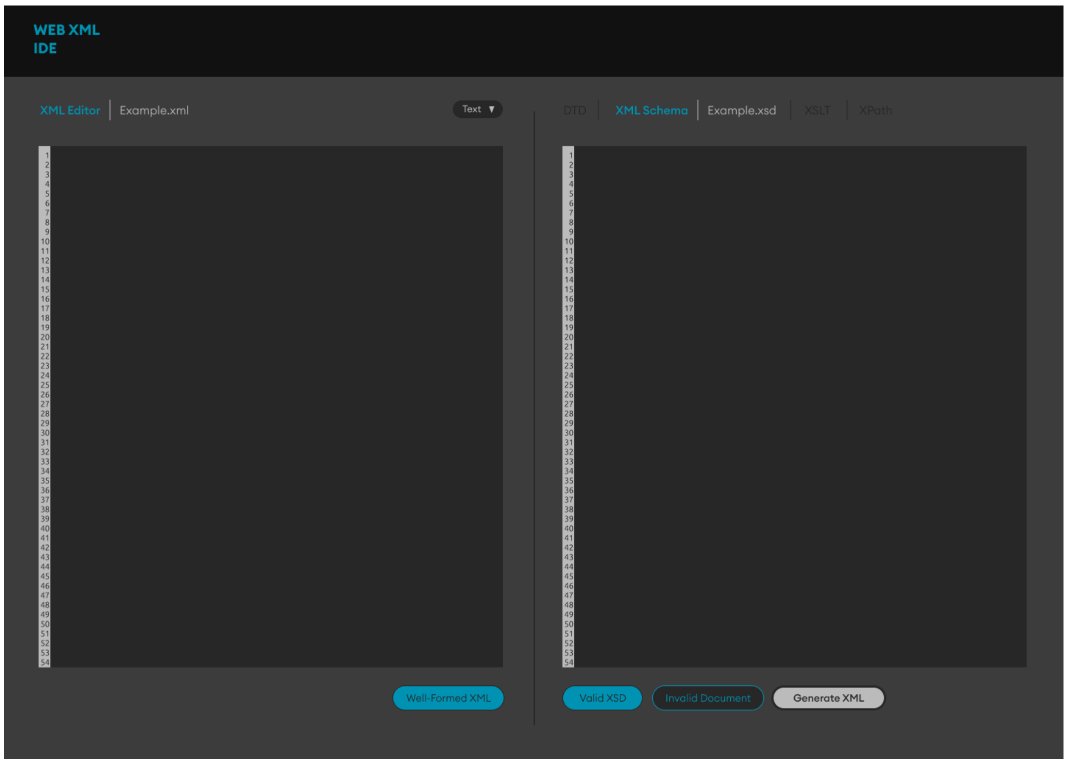


Figure 17: Mockup XSD Editor and Tools

Since the components of the XML side of the web-app have been described previously, let's focus on the components on the XSD workspace:

WORKSPACE SELECTOR - In this case, the XSD workspace is active but the selector allows the user to change workspaces in just one click.

FILE NAME - Allows the user to define a name for the XSD file.

XSD CODE EDITOR - Code Editor where the XSD is defined.

XSD VALIDATOR - Checks if the defined schema in the code editor is valid.

DOCUMENT VALIDATOR - Checks if the XML Document on the XML Code Editor is valid against the defined XML Schema.

GENERATOR - Takes the specified DTD and generates an XML Document that is valid against the same XML Schema Definition, with random values for it's elements.

4.3.4 *XSLT Editor and Tools*

The mockup for the XSLT Editor and Tools page of the proposed app is presented in this section. The left side of the screen is taken with the XML Editor described previously and the right side is a XSLT workspace with some components that will be described later.

Figure 18 contains the mockup for the XSLT Editor and Tools page.

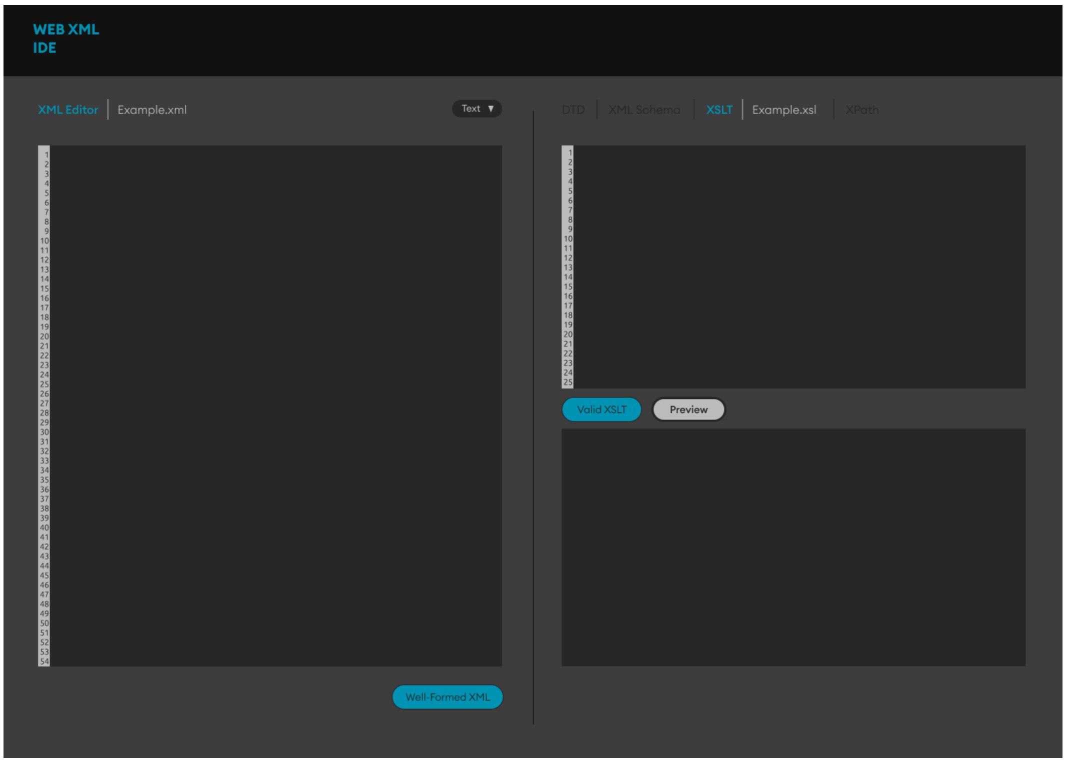


Figure 18: Mockup XSLT Editor and Tools

Since the components of the XML side of the web-app have been described previously, let's focus on the components on the XSLT workspace:

WORKSPACE SELECTOR - In this case, the XSLT workspace is active but the selector allows the user to change workspaces in just one click.

FILE NAME - Allows the user to define a name for the stylesheet definition file.

XSLT CODE EDITOR - Code Editor where the XSD is defined.

XSLT VALIDATOR - Checks if the defined schema in the code editor is valid.

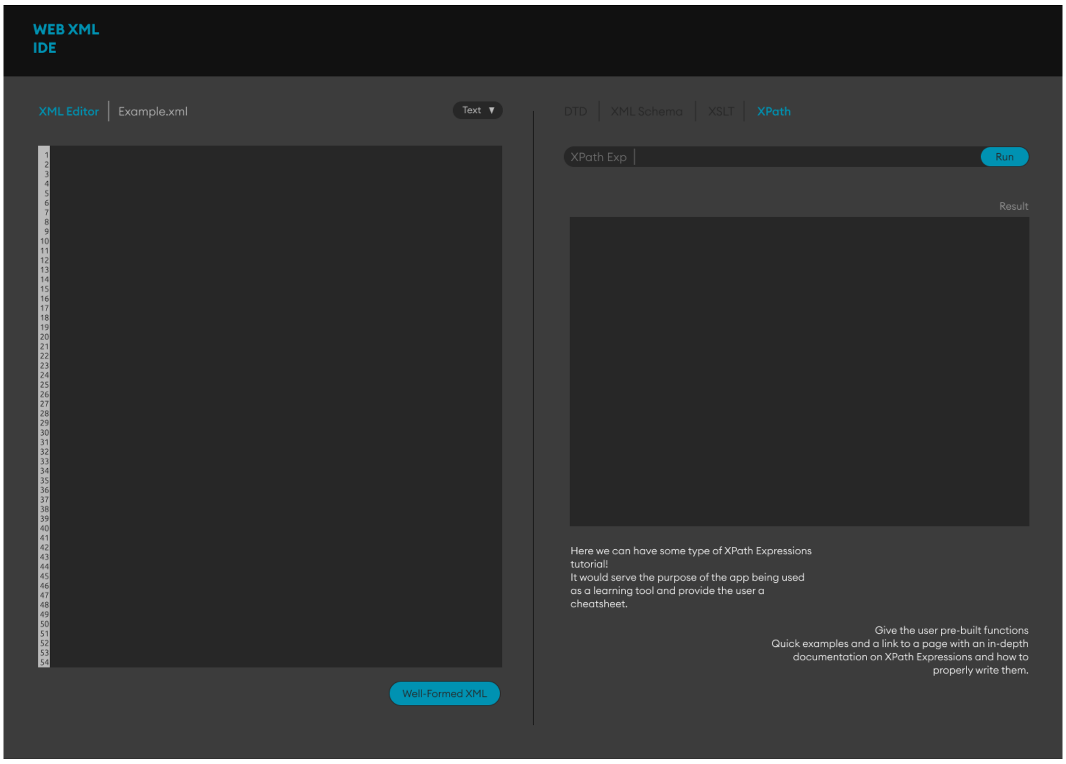
TRANSFORMER - Takes the XML Document defined on the left side of the page and transforms it using the transformations defined in the stylesheet specified in the XSLT Workspace.

OUTPUT PREVIEW - Presents the user with the transformed XML Document in an HTML Preview.

4.3.5 *XPath Tester*

The mockup for the XPath Tester page of the proposed app is presented in this section. The left side of the screen is taken with the XML Editor described previously and the right side is a XPath workspace with some components that will be described later.

Figure 19 contains the mockup for the XPath Tester's page.



Here we can have some type of XPath Expressions tutorial!
It would serve the purpose of the app being used as a learning tool and provide the user a cheatsheet.

Give the user pre-built functions
Quick examples and a link to a page with an in-depth documentation on XPath Expressions and how to properly write them.

Figure 19: Mockup XPath Tester

Since the components of the XML side of the web-app have been described previously, let's focus on the components on the XPath workspace:

WORKSPACE SELECTOR - In this case, the XPath workspace is active but the selector allows the user to change workspaces in just one click.

XPATH EXPRESSION FORMS - This is where the user will write XPath Expressions.

XPATH VALIDATOR AND TESTER - The user won't be able to run the XPath expression if the given expression is not valid. If the expression is valid, the *"Run"* button will text the epxpression on the XML Document defined in the XML Workspace.

XPATH TEST RESULT - Presents the user with the result of applying the queries on the XML Document.

EXTRA - This page could have a simple cheat-sheet with some common predefined XPath functions.

4.4 SUMMARY

The Use Cases presented in Section 4.2 define how the user interacts with the system while the Mockups presented in Section 4.3 will serve as guidelines for building the graphical interface for **WebXMLIDE**.

The next stage in this project is to plan the development of this web application based on the strategies and guidelines defined in this Chapter.

5 DEVELOPMENT

This chapter will explain the decisions made about the technologies and tools used to prototype and implement the proposed Web IDE, as well as the plan for the implementation process and the process itself for the platform and all the proposed requirements and features.

5.1 TECHNOLOGIES AND TOOLS

This section will be about presenting the technologies and tools used to prototype and implement the proposed tool, as well as some components of the web IDE.

5.1.1 *Python Django* - *Prototyping*

When this project first started, **Python Django**[1] was the chosen web development framework due to its popularity, ease of use and rapid development capabilities. It allowed for a quick prototype to be built, which included the implementation of the CodeMirror code editors for XML, DTD, XSD and XSLT, as well as file upload and download, and syntax checking.

However, as the project progressed, it became apparent that some features, such as state persistence when reloading elements on the page (which is key on the UX/UI Design for this Web App), were difficult or impossible to implement using **Django**. This limitation led to the decision to switch to **Vue.js** for the final implementation.

Vue.js offers a range of benefits over **Django**, including *vuex* and *vuex-persistedstate* for state management, which was crucial for the project's requirements. **Vue.js** also provides an easier way to pass information between components, which simplifies the development process.

In conclusion, while **Django** was a suitable choice for prototyping the project, its limitations regarding state persistence and data management led to the decision to switch to **Vue.js** for the final implementation. The use of **Vue.js** and its associated tools allowed for a more

1 https://www.djangoproject.com/

efficient and modular development process, resulting in a more robust, easy-to-maintain and feature-rich Web IDE for XML.

5.1.2 *Vue.js - Development Framework*

Due to its many advantages, **Vue.js**[2], a well-known JavaScript framework, was selected for the Web IDE final implementation.

Using **Vue.js**'s ability to maintain state efficiently through *vuex* and *vuex-persistedstate* is one of its main advantages. This capability is essential for the online IDE since it enables the application to have a constant state even as the user interacts with the different components it offers without ever losing the work that has already been put into place.

Vue.js is also a very flexible and easy-to-use framework as it offers a straightforward and understandable syntax for creating components and controlling how they interact between them.

This feature is very useful in projects like an online IDE for XML where multiple elements (such as syntax checkers, validation tools, different code editors, etc.) need to be connected and communicate with each other.

Additionally, **Vue.js** provides quick and responsive user experience. This is essential when it comes to an IDE since customers demand a fluid editing experience. All of this is made possible via **Vue.js**'s reactive and dynamic user interface, which can adapt and react to user input quickly.

Finally, the availability of tools and libraries that enhance the framework also played a role in the choice to use **Vue.js**. For example, the addition of **CodeMirror** to **Vue.js** made it possible to build high-quality code editors for XML, DTD, XSD and XSLT, and the integration was very easy because **CodeMirror** is a native JavaScript library. There is a wide range of JavaScript libraries (most of them open-source), which made it easy to find solutions for some of the problems faced during the development stage of this project.

In conclusion, **Vue.js** is a strong and adaptable framework that works well for creating web IDEs. It is the best option for a project like the proposed application because of its ability to manage state, offer a responsive UI, and integrate with a wide variety of libraries.

5.1.3 *Express & Node.js - Back-end Server*

There are some JavaScript libraries available for working with XML, but many of them are not compatible with client-side applications (like **Vue.js** applications). A **Node.js**[3] back-end

2 https://vuejs.org/, last accessed 07/08/2023
3 https://nodejs.org/en, last accessed 07/08/2023

server solves this issue by offering a wide range of server-side JavaScript libraries for working with XML, DTD, XSD, XSLT or XPath.

When it comes to choosing a web framework for the **Node.js** server, **Express**[4] is a popular and widely-used choice. It's a minimal and flexible web framework with a simple and intuitive API for handling HTTP requests and responses, as well as support for middle-ware and routing.

5.1.4 *CodeMirror 5 - Web Code Editor*

CodeMirror 5[5] is a powerful and flexible JavaScript library that provides an easy-to-use interface for building code editors on web applications. The library is open-source and has a strong and active community that regularly contributes to its development and maintenance.

One of the key reasons for choosing **CodeMirror** to implement the code editors for XML, DTD, XSD and XSLT in this project is its compatibility with Vue, which is also written in JavaScript. This ensured seamless integration and easy maintenance of the source code. Furthermore, **CodeMirror** has an intuitive and well-documented API that made it easy to customize and extend the editors to meet the project requirements.

In addition to its compatibility and ease-of-use, CodeMirror offers several benefits over other solutions of the same kind (like Ace[6] by Cloud9 IDE and Mozilla, Monaco[7] from Microsoft, Quill[8], Froala[9] or TinyMCE[10]). For instance, **CodeMirror** is highly customizable, with support for a wide range of languages and syntax highlighting. This makes it easy to build code editors for various programming languages and web technologies. In this project, the code editor has to support both XML and DTD syntax, and **CodeMirror** does it very successfully.

Another key advantage of **CodeMirror** is its performance. The library is optimized for speed and can handle large files with ease, making it suitable for use in web-based IDEs. It also offers features like auto-complete, code folding, among others, which enhance the user experience and increase productivity.

In summary, **CodeMirror** is an excellent choice for building code editors for web IDEs. Its compatibility with JavaScript frameworks (like **Vue.js**), customizable features, high performance and active community make it an obvious choice for this project.

4 https://expressjs.com/, last accessed 07/08/2023
5 https://codemirror.net/5/, last accessed 07/08/2023
6 https://ace.c9.io/, last accessed 07/08/2023
7 https://microsoft.github.io/monaco-editor/, last accessed 07/08/2023
8 https://quilljs.com/, last accessed 07/08/2023
9 https://froala.com/wysiwyg-editor/, last accessed 07/08/2023
10 https://www.tiny.cloud/, last accessed 07/08/2023

5.1.5 ***Libxmljs2*** *- XPath Query Tester*

Libxmljs2[11] is an open-source **LibXML2**[12] **Node.js** Wrapper. The library that appears on the next section offers features focused on very specific use cases while **Libxmljs2** offers solutions to the most common use cases.

In this project, this library is being used to serve as an XPath Tester since **node-libxml** is very limited when working with XPath, while **Libxmljs2** has a very complete and intuitive API when it comes to working with XPath.

5.1.6 ***node-libxml*** *- Schema Parser and Validator*

node-libxml[13] is another open-source LibXML2 Node.js Wrapper. While **Libxmljs2**[14] was designed and thought to accommodate the most common use cases, **node-libxml** offers more specific use cases like validating an XML document against one or more DTDs or XSDs, among others.

This library offers a simple and easy-to-understand API and works perfectly according to this project's requirements on working with DTDs and XSDs.

5.1.7 ***xmldom*** *- Server-side XML Parsing*

xmldom[15] is a JavaScript implementation of W3C DOM for **Node.js**. It provides useful APIs like converting an XML String into a DOM Tree; creating, acessing and modifying a DOM Tree; and serializing a DOM Tree back into an XML String, that are present in modern browsers to the **Node.js** runtime.

This library tries to stay as close as possible to the various standards when it comes to fixing bugs and implementing features. What this means is that the API will work just like the W3C DOM in modern browers, making it easy to find documentation, code examples and bug fixes.

11 `https://github.com/marudor/libxmljs2`
12 `https://github.com/GNOME/libxml2`, last accessed 07/08/2023
13 `https://github.com/MatthD/node-libxml`
14 `https://github.com/marudor/libxmljs2`
15 `https://github.com/xmldom/xmldom`

5.1.8 ***Trang*** *- DTD to XSD Converter*

Trang[16] is a command-line tool that translates from one schema language to another in a way that preserves all the aspects of the input schema including definitions, annotations and comments (Clark, 2008).

This command-line tool was used on the **Node.js** server for two main reasons:

1. Produces high-quality results when converting DTDs to XSD Schemas;
2. Lack of native JavaScript libraries that support converting DTDs to XSD Schemas.

Trang is used by running a *.jar* file that takes an input file containing the DTD specification as an argument and creates a second file with the corresponding specification in XSD.

5.2 DEVELOPMENT PLAN

The planning stage for the development of this project is described in this section. This stage includes planning Vue Components and Architecture, as well as revisiting and taking a more in-depth look at the project's requirements.

5.2.1 *Planning Components*

The plans for the Vue Components and UI/UX Elements are presented in this section.

The **Vue.js** framework works with **components**, that allow the developer to divide the UI into smaller, re-usable pieces of code and think of each of these smaller parts individually (Porter, 2022). With this and the mockups presented in chapter 4.3, the plan for the Vue Components will take into consideration the proposed UI and divide it into sections and later into Vue components and HTML Elements.

In the following Figures, UI Parts and components are differentiated by the font weight. UI Parts are identified by having a light font while components are identified with a bold font.

HOME PAGE

The Home Page for the proposed application will have four components: **Navbar**, **XML-Side**, **CompanySide** and **Footer** (see Figure 20).

16 `https://relaxng.org/jclark/trang-manual.html`, last accessed 07/08/2023

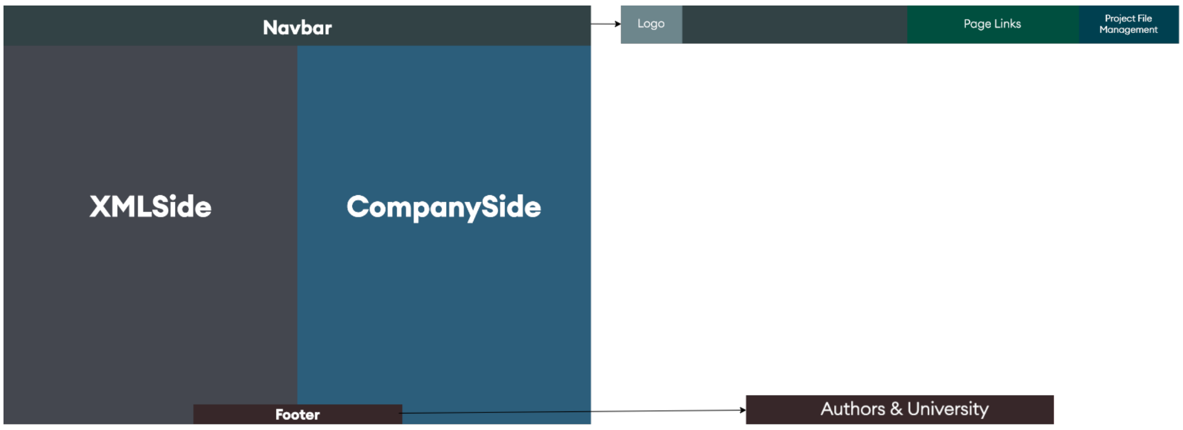


Figure 20: Home Page

The **Navbar** will have three sections within the component. One for the application logo, one for page links and the last for project file management (project upload and download buttons).

The **Footer** will have just one section containing information about this projects authors.

XMLSIDE

The **XMLSide** component will be divided into three sections: *XML-TOP*, *XML-MIDDLE* and *XML-BOTTOM* (see Figure 21).

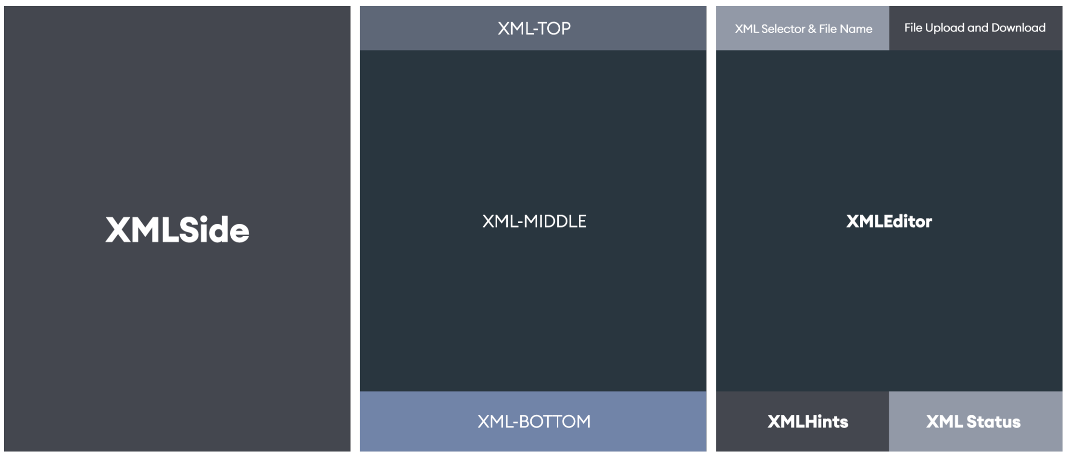


Figure 21: XMLSide

The *XML-TOP* section will contain two sections: *XML Selector & File Name* and *File Upload and Download*. The first will contain a label displaying "XML" and a text input where the user can write the desired file name for the XML Document. The second section will contain two buttons, allowing the user to upload XML files to the editor or download the current file to the file system.

The *XML-MIDDLE* section will contain one single component called **XMLEditor**. This is where the XML Code Editor (built with **CodeMirror**) will appear.

The *XML-BOTTOM* section will contain two components: **XMLHints** and **XMLStatus**. The first will provide shortcuts for the XML Code Editor to the user, while the second will present to the user if the XML Document is Well-Formed or not (and why it isn't).

companyside

The **CompanySide** component will be divided into two sections: *Company-TOP* and *Company-MIDDLE-BOTTOM* (see Figure 22).

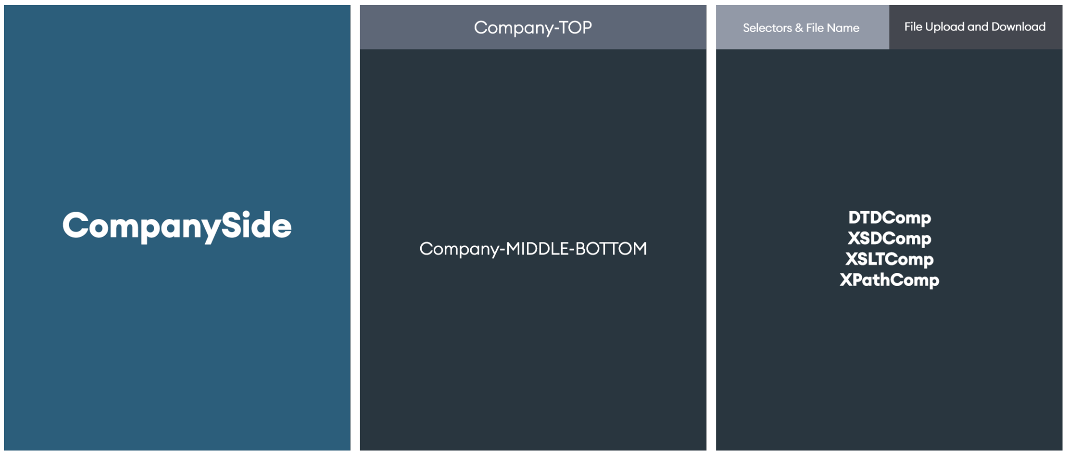


Figure 22: CompanySide

The *XML-TOP* section will contain two sections: *Selectors & File Name* and *File Upload and Download*. The first will contain a selection menu for the user to choose between working with DTD, XSD, XSLT or XPath and a text input where the user can write the desired file name for the DTD, XSD or XSLT file. The second section will contain two buttons, allowing the user to upload DTD, XSD or XSLT files to the editor or download the current file to the file system. Both file name input and file management sections are disabled for the XPath editor.

The *XML-MIDDLE* section will contain one of four components (depending on what the active selector is on the *Company-TOPs Selector* section): **DTDComp**, **XSDComp**, **XSLTComp** or **XPathComp**. Each of these is described in the next sections.

dtdcomp & xsdcomp

The **DTDComp** and **XSDComp** components will be presented together as they are very similar. Both of these components contains three components within itself: an **Editor** component (**DTDEditor** and **XSDEditor**), a **Status** component (**DTDStatus** and **XSDStatus**) and an **Actions** component - **DTDActions** and **XSDActions** (see Figure 23).

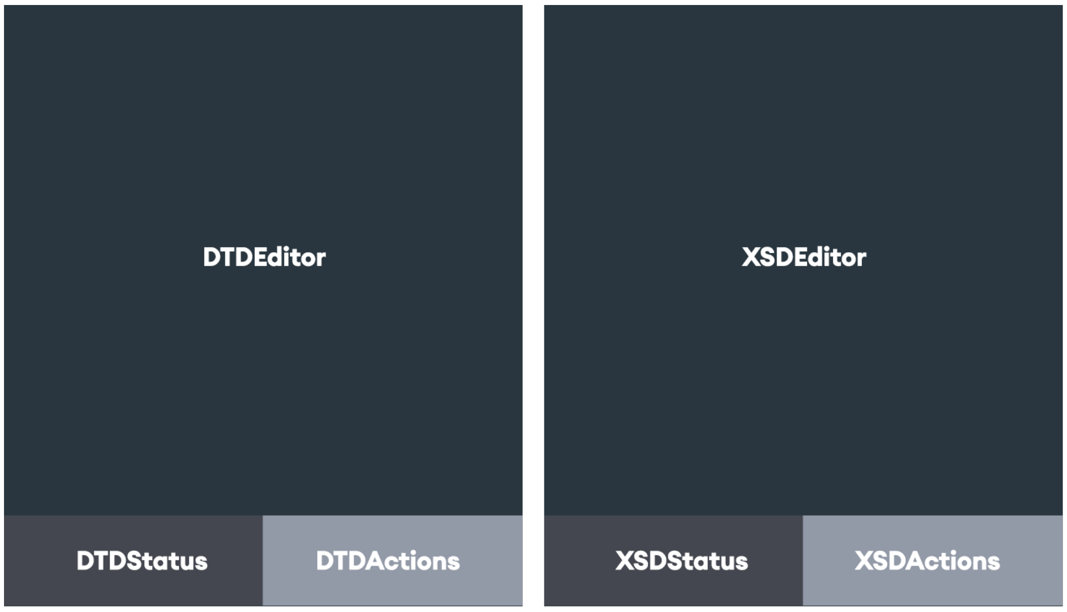


Figure 23: DTDComp & XSDComp

The three components contained in both **DTDComp** and **XSDEditor** have the following purposes:

DTDEDITOR AND XSDEDITOR - This is where the code editors will be mounted on.

DTDSTATUS AND XSDSTATUS - This is where the system will inform the user if the given DTD/XSD is valid and if the XML Document on the **XMLSide** is valid against the given DTD/XSD.

DTDACTIONS AND XSDACTIONS - The **DTDActions** component will give the user the options to convert the given DTD to a XSD file and to generate an XML Document (with random data) based on the given DTD and valid according to it. The **XSDActions** component only offers the latter.

XSLTCOMP

The **XSLTComp** will contain four components: **XSLTEditor**, **XSLTStatus**, **XSLTActions** and **XSLTPreview** (see Figure 24).

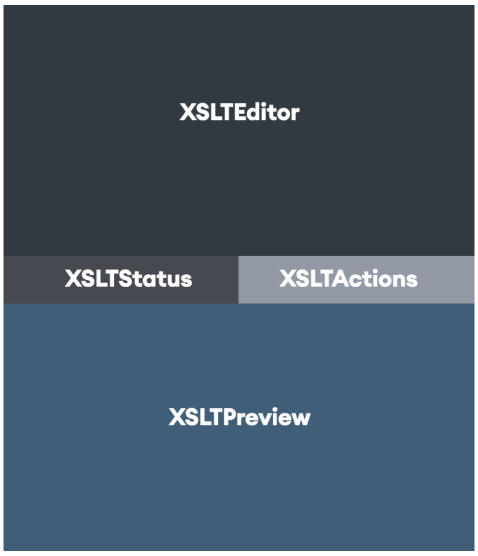


Figure 24: XSLTComp

The **XSLTEditor** is where the CodeMirror code editor will appear. The **XSLTStatus** informs the user if the given style sheet is valid. The **XSLTActions** allows the user to update the content presented in **XSLTPreview**, which shows the result of applying the transformations defined in the XSLT file to the XML Document in the **XMLSide**.

XPATHCOMP

The **XPathComp** will contain four components: **XPathEditor**, **XPathOutput** and **XPath-Hints** (see Figure 25).

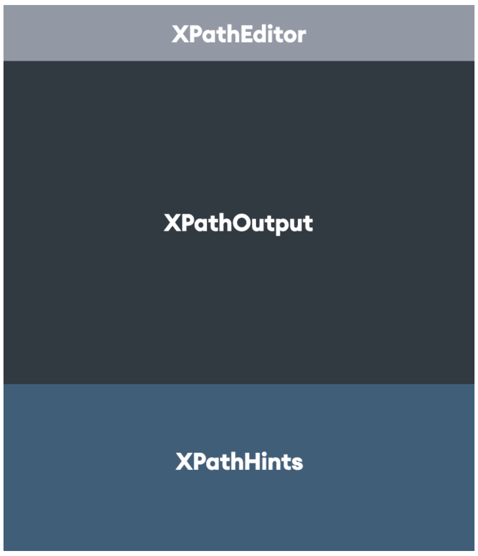


Figure 25: XPathComp

The **XPathEditor** component will be a text input where the user can write XPath expressions and run them on the XML Document in the **XMLSide**. The **XPathOutput** will present the user with the result of running the given XPath expressions on the XML Document.

5.2.2 *Final Architecture for the Web Application*

The block diagram in Figure 26 showcases in detail the final architecture for the Web Application.

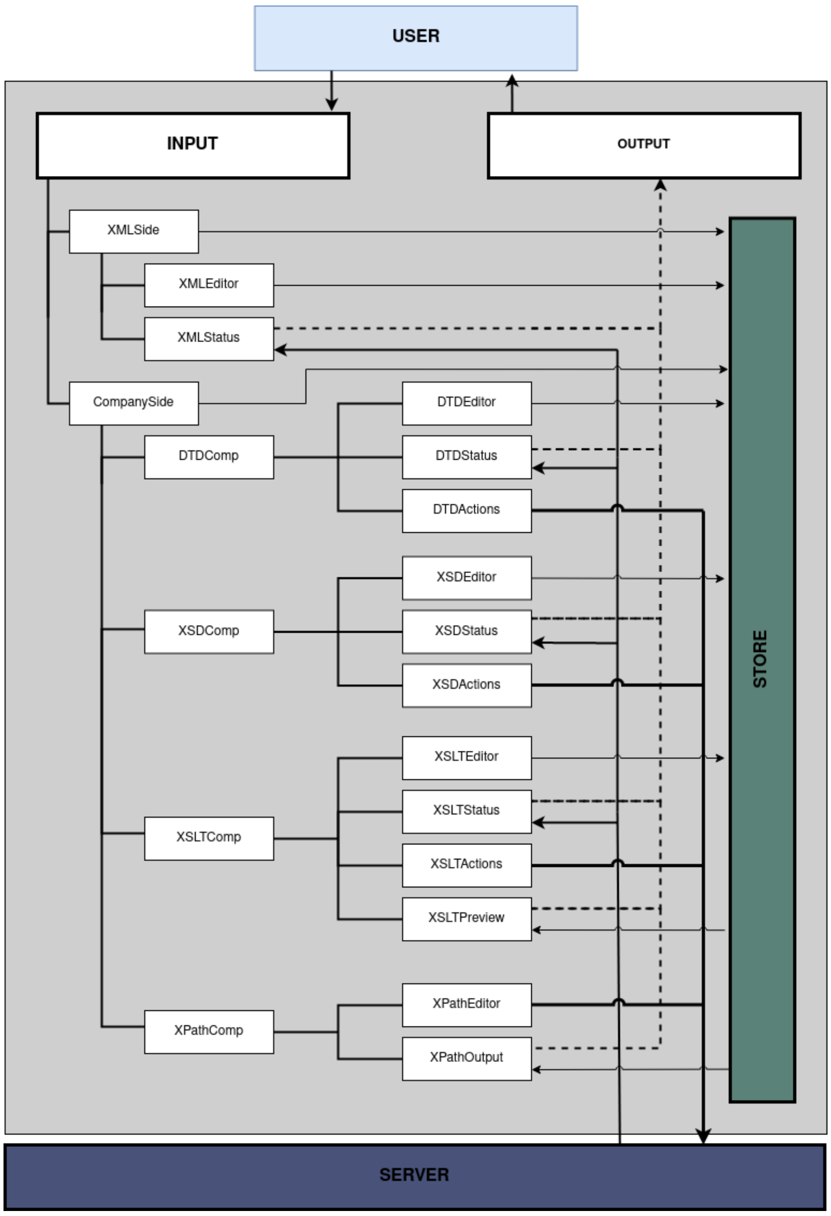


Figure 26: Detailed Architecture for the Web Application

As can be observer in Figure 26, the showed diagram contains 6 different blocks:

USER - The actor will interact with the system by providing an input and receiving an output in return.

INPUT - The input provided by the user can be XML, DTD, XSD or XSLT Code, as well as XPath expressions and file names.

OUTPUT - The output given to the user can differ depending on the given input. This output may consist of files (if the user wants to download the files), syntax or validation errors, XSLT previews or the results from running XPath queries.

FRONT-END APPLICATION - The front-end application block represents the Vue App and all its components. This block shows the structure of the components inside the application.

BACK-END SERVER - The back-end server block represents the Node.js server used to process code and execute the XML-related features.

STORE - This block represents the *vuex-persistedstate* object which will act as a state recorder and loader.

5.2.3 *Requirements and Features*

This section presents a deeper analysis and specification of the system's features presented in Section 4.2. It also contains the specification for technical requirements that are not specific to one of the previously mentioned features but are key to the users quality of life.

5.2.3.1 Technical Requirements

The technical requirements for this project represent the features needed to fulfill the functional objective for the proposed application. Things like code editors and their corresponding features, components used to provide information to the user, buttons used to trigger server-side actions, etc.

CODE EDITORS

CodeMirror provides amazing and easy-to-build code editors for editing XML, DTD, XSD and XSLT files. This library also has a few add-ons and plugins that could be used in this project to fulfill the goal of making these code editors beginner-friendly and easy-to-use. For the XML-syntax-based languages (XML, XSD and XSLT), the code editors should offer syntax highlighting, code suggestions and auto-completion. All of these are easily implemented using **CodeMirror**'s add-ons. When it comes to DTDs, however, there is no support for code suggestions or auto-completion, but syntax highlighting is still available at a very high level in quality.

STATUS CHECKERS

Status should be visible to the user at all times and be responsive to the user's input. With this in mind, we can define we want to know the status of the editor's files (if they're well-formed, if they're valid, and so on) as the user is typing, meaning the status should be checked at every change made to the documents. The status we want to present the user are the following:

FILE NAME VALIDITY - The names for the files in the editors should not contain certain characters and should always have the right file extension at the end of the file name. The testing for this status will be made client-side by the **Vue.js** application.

XML WELL-FORMED - This status checker will take the XML code in the respective editor and send a request to the **Node.js** server, which will respond back with information regarding to the well-formed status of the provided code. If the given XML is not well-formed, it will also send a list of errors and present them to the user.

DTD, XSD & XSLT SYNTAX CHECKERS - These syntax checkers will inform the user if the given DTD, XSD or XSLT code is syntactically correct. The parsing process will be done in the **Node.js** server by sending it a request the given code and checking if any errors occur during the parsing process. It is important to inform the user if the syntax is correct but it is also important to be able to present the user the errors (if they exist) found in the parsing process.

XML VALIDITY CHECKER AGAINST DTD/XSD - This validation process will be made in the **Node.js** server by sending it a request with the XML code, the DTD or XSD code and the given DTD or XSD file name. The validation can have many different outcomes: XML and DTD/XSD are not linked, the link between the two files is incorrect, XML does not follow a correct DTD/XSD, XML is not well-formed (so it can't be validated), DTD/XSD is not syntactically correct, and finally, XML is valid against the given DTD/XSD. All of these possible errors should be visible to the user so that fixing these issues becomes easier.

ACTIONS

In this specific case, actions should be considered as the events the user can trigger from the system's UI.

There are four actions associated with the XML work environment discussed in this report that should be considered in the development stage for this project:

CONVERT DTD TO XSD - When this action is triggered, the system should take the DTD code in the DTD Editor and run it through **Trang** (see section 5.1.8) as the input argument, and should get as an output the converted XSD file. Taking the resulting XSD file, the system should change the displayed editor from DTD to XSD, changing the content of the XSD editor to the content of the output file and the filename to the resulting filename from the conversion (which will be the file name for the DTD, replacing the *.dtd* extension with the *.xsd* extension). During this process, the DTD should be preserved, but the state persistence will be further discussed in a following section.

GENERATE XML FROM DTD OR XSD - When this action is triggered on either the DTD or XSD page, the system should take the DTD or XSD code and generate XML code that is valid against the given DTD or XSD, and well-formed. After generating the code, it should change the content of the XML editor to the resulting XML code.

GENERATE XSLT PREVIEW - This action will take the code in the XSLT editor and the XML code in the XML Editor and generate an HTML preview that will be presented to the user. If there are any errors found in either the XSLT or XML Code, these errors will be shown instead of the HTML Preview.

RUN XPATH EXPRESSIONS - This action will take the XML code and the given XPath Expression and test the expression on the document, returning the results or the errors that may occur during the testing process to the user.

FILE MANAGEMENT

Last but not least in this section, the user must have the ability to manage files.

FILE UPLOAD AND DOWNLOAD - Every code editor (XML, DTD, XSD and XSLT) should allow for the user to upload a file replacing the value in the file name with the name of the uploaded file, and replacing the content in the editor with the content from the uploaded file. The user should also be able to download the content in each editor to its own file system. This download procedure should take the file name defined in the editor and set it as the name for the downloaded file.

PROJECT UPLOAD AND DOWNLOAD - The user should be able to upload or download a project as a whole (meaning, every piece of content in each editor should be downloaded or uploaded as one). This process should work as a *.zip* upload and download. When uploading, the user should select a zip containing the project files inside a *src* folder and an XML manifest that should inform the system on which of the files corresponds to the each of the systems components. For example, the manifest can state that the uploaded project only has files for the XML and DTD editors, discarding any other files uploaded with the*.zip*. When downloading a project, the system should show the users with a forms allowing them to select which files they want to download (XML, DTD, XSD and/or XSLT) and a name for the project. When the user is done choosing which files to download and the project name, the system should download a *.zip* folder containing the XML manifest and a *src* folder containing the files specified in the manifest.

5.2.3.2 Quality of Life Improvements

This section presents a few features that are not necessary to the system's functionality, but are key to improving user experience and achieving the goals for this project to make the IDE easy-to-use and beginner-friendly.

STATE PERSISTENCE

State persistence means the system should maintain the user's work until the moment the user changes that given state.

Let's say the user is working on a DTD, writing DTD in the given editor and decides to test some XPath Expressions or go back to the XSD editor and revisit a previously defined schema. The user should never lose the content on the DTD component when changing the active component.

This also applies to when the user closes the tab in which the IDE is open. When the user closes the IDE, it should maintain the state so that, when the user returns, the previous work is still available.

This state persistence should be applied on the content of the different code editors, the file name inputs for every component, the XPath expressions and results and the XSLT preview.

Reducing the amount of work the user has to put into place every time the user enters or leaves the page is key to making the user experience enjoyable. State persistence is a viable solution to fix this problem, especially considering *vuex* and *vuex-persistedstate*.

DOCUMENTATION PAGE

The system should also offers the user with a Documentation Page with links to detailed tutorials and documentation guides on XML, DTD, XSD, XSLT and XPath.

KEYBOARD SHORTCUTS

In each code editor, the user should have the ability to use some shortcuts to reduce the amount of work and mouse usage when working with the IDE. Some of the actions that should be allowed with these shortcuts are:

1. Adding XML Declaration (XML-syntax-based languages)
2. Get a Code Suggestion
3. Auto-Indent
4. Code-Folding/Unfolding

It is important to notice that **CodeMirror** offers the *special-keys* feature to enable shortcuts and defining actions for these shortcuts, which makes defining these shortcuts much easier.

5.3 DEVELOPMENT PROCESS

This section presents the results from the development process following the planning process described in the previous section.

This section presents the final system architecture as well as the implementation for the projects proposed features.

5.3.1 *Features Implementation*

CODE EDITORS

The four proposed code editors for XML, DTD, XSD and XSLT have all been built using **CodeMirror 5**.

The XML, XSD and XSLT code editors were built using **CodeMirror**s XML mode, which offers high-quality syntax highlighting to XML-syntax based languages. These code editors also have the ability to provide suggestions. As it stands, the editor can only provide information on the tag the user is supposed to close next, which is still a big help when writing XML Documents, Schemas or XML stylesheets, especially for non-experienced users. Code suggestions on **CodeMirror** is Schema-based. This means the code suggestion mechanism put into place in the XML Editor could be further developed to match the defined schema on the XSD or DTD editors. These editors also allow for code folding and auto indentation.

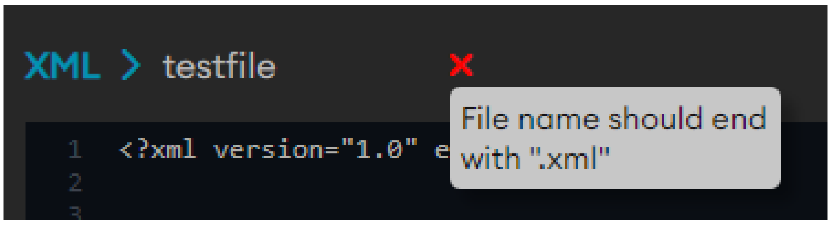
The DTD Editor was built using **CodeMirrors** DTD mode, which is a bit incomplete but still good enough to allow the user to quickly understand the DTD code. Code folding and auto indentation are also available but code suggestions are not.

Overall, using the resulting code editors feels intuitive and offers a smooth user experience. This feature was successfully accomplished.

STATUS CHECKERS

Status checkers were implemented using a combination of Front-End and Back-End operations reacting to the users input as it happens. This makes it so that every state of the users input is tested and the result of checking for its status is shown on every input change.

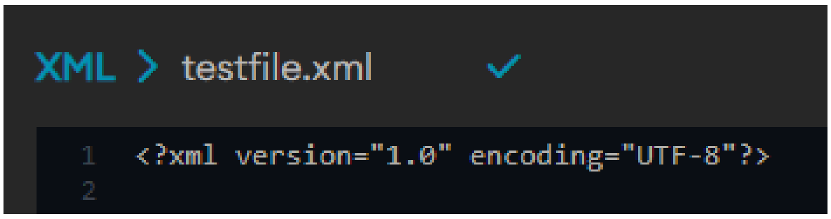
FILE NAME VALIDITY - Checking if the file contains the right file extension and contains any invalid filename characters was accomplished using **regular expressions** on the Client-side application (i.e., the Vue App). If the file fails this validity test, the user will be informed on the current problem on the input filename. A quick example is shown in Figure 27.



```
<?xml version="1.0" e
```

Figure 27: Invalid File Name Warning

If the user inputs a valid filename that ends with the correct file extension, that information is also shown, helping less experienced users to always take this into consideration when using any of the available editors, as shown in Figure 28.



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

Figure 28: Invalid File Name Warning

XML WELL-FORMED - When the user changes the code in the XML Editor, the Vue app sends a POST request to the Node.js server on route *'/xml-wellformed'* with the current XML code. When receiving the request, the server will take the XML code and pass it on to the XML parser from **node-libxml** (see Subsection 5.1.6). This parser can take XML

Code input two different ways: as a file (with a path) and from String. Since the server is receiving the XML code as a string, it made sense to avoid creating temporary XML files and just pass the received XML code to the parser. Once the parser is finished, it can either fill an array with the errors it found or leave it empty if it finds none. The server will send back a response containing two body fields: *wellFormed* and *errors*. The first is a simple flag that indicates if the passed XML code is well-formed or not. The second is the result of taking the parsing errors found by **node-libxml** and creating an array of strings with each error message as a different element of the array. The Vue app will then take this response and present it to the user as can be seen in Figures 29 and 30.

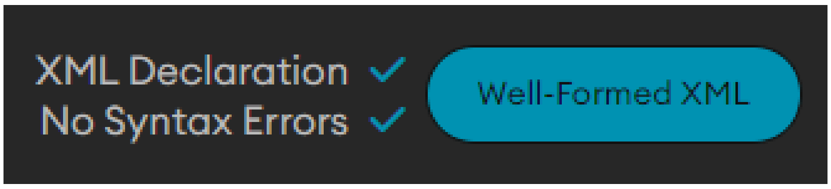
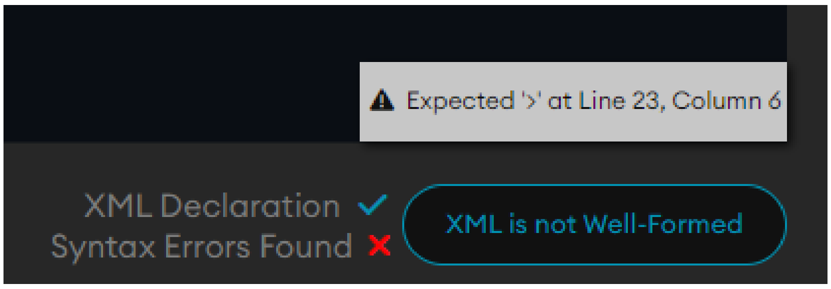


Figure 29: "XML is Well-Formed" Indicator



Expected '>' at Line 23, Column 6

Figure 30: "XML is not Well-Formed" and Errors Indicator

DTD, XSD & XSLT SYNTAX CHECKERS - The idea behind these syntax checkers is the same as it was for the previous XML status checker. The Vue app sends requests to the Node.js server on every change the user makes on the editors, making sure every state of the code is evaluated. The requests are made to the server on routes *'/validate-dtd'*, *'/validate-xsd'* and *'/validate-xslt'*. The first two routes serve more than one purpose as these are also the routes used to check if the XML is valid against the given DTD or XSD. For parsing the DTD, the server uses the parser from **node-libxml**, just like it did with XML. The main difference is that the parser will only admit DTDs as files and the file is required to be loaded into the object. To accomplish this requirement, the server uses a *'temp'* folder (creates one if needed) to create a temporary file with the DTDs filename and pass it to the parser. When loading the DTD from the path to the object, the parser will fill an array with any errors it finds or leave it empty if none are found. With this information, we can know if the parser considers the DTD code to be correct or not and, if it isn't, what are the errors that need to be fixed. This is the information the status indicator will be using to let the user know if the defined DTD is correct or not. The server will send this information to the front-end application, along with some more information about the validity of the XML code against the given DTD (which will be explained further in this report). Figure 31 and 32 illustrate how the information is presented to the user.

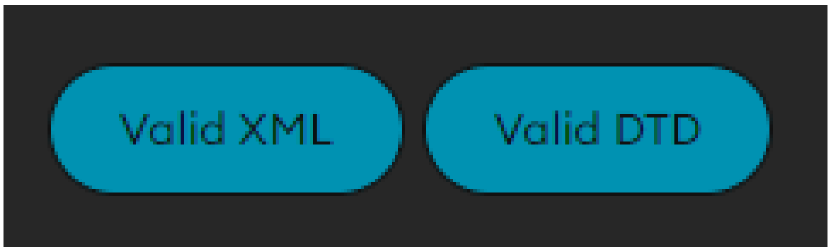


Figure 31: "Valid DTD" Indicator

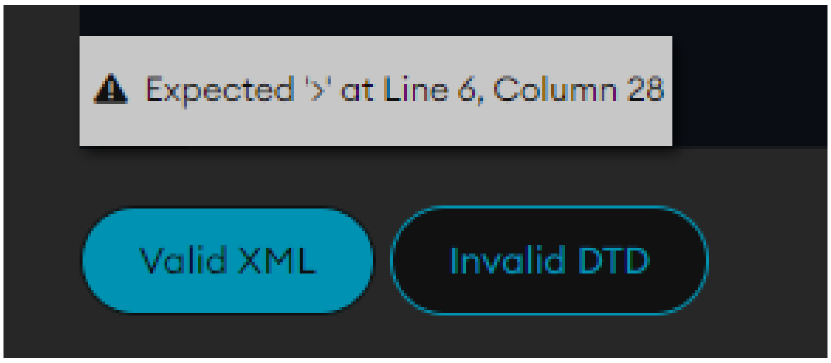


Figure 32: "Invalid DTD" and Errors Indicator

Both XSD and XSLT work exactly the same way as the XML Status Checker described previously and won't be explained to avoid repetitiveness.

XML VALIDITY CHECKER AGAINST DTD/XSD - When the front-end app sends the POST requests on every change the user makes on the editors, the server will check if the given DTD or XSD Code is correctly written and then will test the given XML Document against the given DTD or XSD using **node-libxml**s tester. The first test it performs is checking if the given XML Document is well-formed since an XML Document needs to be well-formed first to be later considered valid against a DTD or Schema. Then, it checks if the DTD or XSD and XML files are correctly linked using the *DOCTYPE* block or *xsi:schemaLocation* attribute on the schema element of the given XML Document. If the files are properly linked, the server will check if the XML Document is valid against the given DTD or XSD. If it isn't, it will send an array with error messages indicating why the validation process failed. If the Document turns to be valid, the server will just send an empty array back to the front-end application and a *boolean* field stating the validation process returned true. Figure 33 illustrates the notification sent to the user when the XML document and the DTD are not linked.

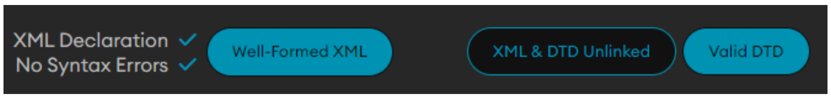
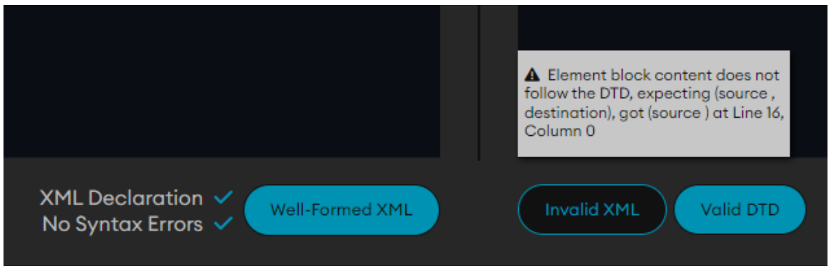


Figure 33: "Unlinked XML & DTD" Indicator

Figure 34 shows how the user will be notified if the XML and the DTD files are properly linked but XML is invalid when tested against the given DTD.



Element block content does not follow the DTD, expecting (source , destination), got (source) at Line 16, Column 0

Figure 34: "Invalid XML" and Validation Errors Indicator

Last but not least, Figure 35 shows how the user knows the XML File and DTD File are properly linked and the XML is valid against the given DTD.

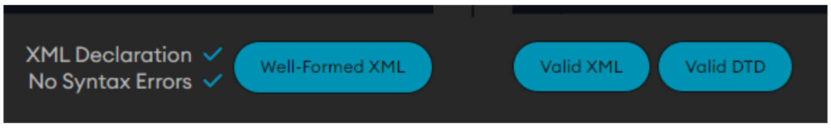


Figure 35: "Valid XML" Indicator

The UI on the XSD component is similar to the one on the DTD component and will not be shown to avoid repetitiveness.

ACTIONS

The proposed actions were implemented using a combination of back-end libraries such as **Trang** and Front-End features such as the browsers **DOMParser** and **XSLTProcessor**.

One of the proposed actions didn't get implemented as there were no real solutions to generate XML Documents based on DTDs or XML Schemas. Developing a schema-based XML-generator would solve this problem and should be considered as future work.

CONVERT DTD TO XSD - When the user clicks the "Convert to XSD" button on the DTD Component, the front-end application will send a POST request to the server containing the DTD Code and the current filename. The server will take this input and running **Trang** on a child-process with the path to a temporary file with the DTD filename and content and the path to a temporary XSD File with the same name as the DTD (replacing the file extension). Once **Trang** is done converting, the server will read the content on the *.xsd* file and send it back to the front-end app that will change the active component from DTD to XSD, as well as place the returned code in the XSD Editor and the new file name in the file name input field.

GENERATE XSLT PREVIEW - When the user clicks the "Generate Preview" button on the XSLT Component, the front-end app will take the XSLT code in the editor and the XML Document on the XML Editor and use the browser's native **DOMParser** and **XSLTProcessor** to parse both pieces of code. It starts by creating parsed documents using **DOMParser**, then it imports the XSLT Document as a Stylesheet to the **XSLTProcessor**. After importing the stylesheet, the **XSLTProcessor** will transform the XML Document. Using **XMLSerializer** we can extract the transformed document as HTML Code which will be presented to the user in the output field in the XSLT Component. The user also has the ability to clear the output field.

RUN XPATH EXPRESSIONS - When the user clicks the "Run" button on the XPath component, the front-end app will send a POST request to the server containing the XML Document and the input XPath Expression on route *'/run-xpath'*. The server will receive the XML Document and check if it is well-formed or not. If the document is not well-formed, the server will respond to the request with that information to be presented to the user. If the document is well-formed, it will use **libxmljs2**5.1.5 to parse and create a Document Tree from the XML Document. It will then test the XPath Expression on the created Document Tree and the result can be one of four things: *null*, which means the XPath Expression was invalid; *an empty string*, which means there were no matches to the input query; *an array of matches*, which means the tester found results to the expression; and a *non-empty string* which means the result from the expression is a simple string (when using functions, for example). All of these results are treated to be easy for the user to read and understand the result of running the given XPath expression on the XML Document. The results are sent to the Front-End application as a single string that will be displayed in the output field of the XPath Component.

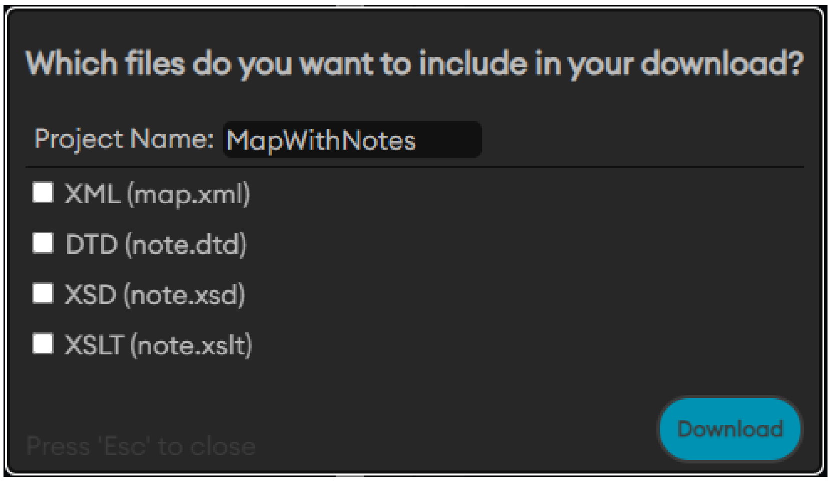
FILE MANAGEMENT

Considering the file upload and download for the different editors are simply HTML input elements that change the content of a state variable in Vue's persisted-state store, there is no need to go into deeper explanations as it was simple to implement this feature for the XML, DTD, XSD and XSLT Editors.

It is now time to explain how the project upload and download features were implmented.

PROJECT UPLOAD - When the user clicks the "Upload" button in the application's top-menu, a file explorer will open, allowing the user to select a *.zip* folder to be uploaded. Once the file is uploaded, the front-end app will look for a filed called *"manifest.xml"* and a *src* folder. If the folder contains both items, the app will take the manifest and see which files contain the desired content for each component and take the files content and name and set the editors content and file names to match the uploaded ones. The user will receive an alert stating the user has to refresh the page to see the changes made. Once the page is reloaded, the uplodade files should be in the corresponding components editor.

PROJECT DOWNLOAD - When the user clicks the "Download" button in the application's top-menu, a forms is displayed with check-boxes that allow the user to select which files are desired to be downloaded. The form also contains a text input field to let the user define a name for the project which will be the name for the downloaded *.zip*. Figure 36 shows the forms presented to the user when downloading a project on **WebXMLIDE**.


Which files do you want to include in your download?

Project Name: MapWithNotes

- [ ] XML (map.xml)
- [ ] DTD (note.dtd)
- [ ] XSD (note.xsd)
- [ ] XSLT (note.xslt)

Press 'Esc' to close

Figure 36: Project Download Form

The application will take the users responses on the forms and create a *.zip* folder containing a *src* folder and the *manifest.xml* file stating every relevant component and file in the project. For example, for the previous form, if the user selected all components (XML, DTD, XSD and XSLT), Example 5.1 would be the resulting *manifest.xml* file.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MapWithNotes">
    <src>
```

```
        <xml>map.xml</xml>
        <dtd>note.dtd</dtd>
        <xsd>note.xsd</xsd>
        <xslt>note.xslt</xslt>
    </src>
</project>
```

Example 5.1: Downloaded Project Manifest

5.3.1.1 Quality of Life Improvements

The implementation the proposed quality of life improvements was successful.

STATE PERSISTENCE

Using *vuex* and *vuex-persistedstate*, it was easy to maintain a persistent state, allowing the user to change pages or even close the app without worrying about losing the current work.

To achieve a total state persistence, the store had to keep track of various things:

1. XML, DTD, XSD and XSLT Code
2. XSLT and XPath Output
3. Flags to determine which of the components editor is active at the moment
4. The filenames for every component

To change the state of these variables, various mutations were defined and used in the components to update the content on any of them according to the user's actions and inputs.

DOCUMENTATION PAGE

The documentation page shows links to tutorials on XML, DTD, XSLT and XPath that are meant to be simple and easy-to-understand as can be observed in Figure 37.

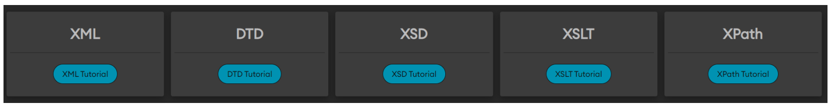


Figure 37: Documentation and Tutorial Links

keyboard shortcuts

Using **CodeMirror**s *special-keys* feature, the user can now perform the following actions:

1. Press Alt+X to add the XML Declaration at the top of the document (XML-syntax-based languages)
2. Press Ctrl+Space to get a code suggestion
3. Press Alt+I to auto-indent at cursor level
4. Press Alt+Q to fold/unfold code at cursor level

5.4 development summary

This Chapter presented the many different technologies used in this project, from prototyping with **Django** (see Subsection 5.1.1), to building the final application with **Vue.js** (see Subsection 5.1.2), and **Node.js** (see Subsection 5.1.3), along with many different JavaScript libraries (see Subsections 5.1.4, 5.1.5, 5.1.6, and 5.1.7), and other utilities like **Trang** (see Subsection 5.1.8).

After presenting the technologies that would be used in this project, it was necessary to take in consideration the Design Mockups from Section 4.3 to plan the **Vue.js Components** to be implemented in the final application (see Subsection 5.2.1). The Use Cases defined in Subsection 4.2 served as the foundation for the System Architecture presented in Subsection 5.2.2. The last thing that was needed was to revisit the list of requirements and provide a more specific definition for each one of the proposed requirements (see Subsection 5.2.3).

After defining Use Cases, Design Mockups, the System Architecture and having specified the required features, the last thing to do was to implement the final application. This process is explained in detail in Section 5.3, providing an in-depth explanation on how the different features were implemented in the final application.

Chapter 6 will present the final product, as well as an analysis on how **WebXMLIDE** compares with the IDEs analyzed in Chapter 3 and the initially proposed application.

6

WEBXMLIDE - FINAL PRODUCT

This chapter contains screenshots of the final product, as well as a summary on what features were implemented and which were not, being postpone for future work.

The final product can be accessed using the following link: https://webxml.epl.di.uminho.pt/.

6.1 SCREENSHOTS

This section contains screenshots of every page in the application, showing how the system responds to an example provided in many W3C Tutorials[1] (see Examples 6.1, 6.2) applied on every component.

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

Example 6.1: Note.xml

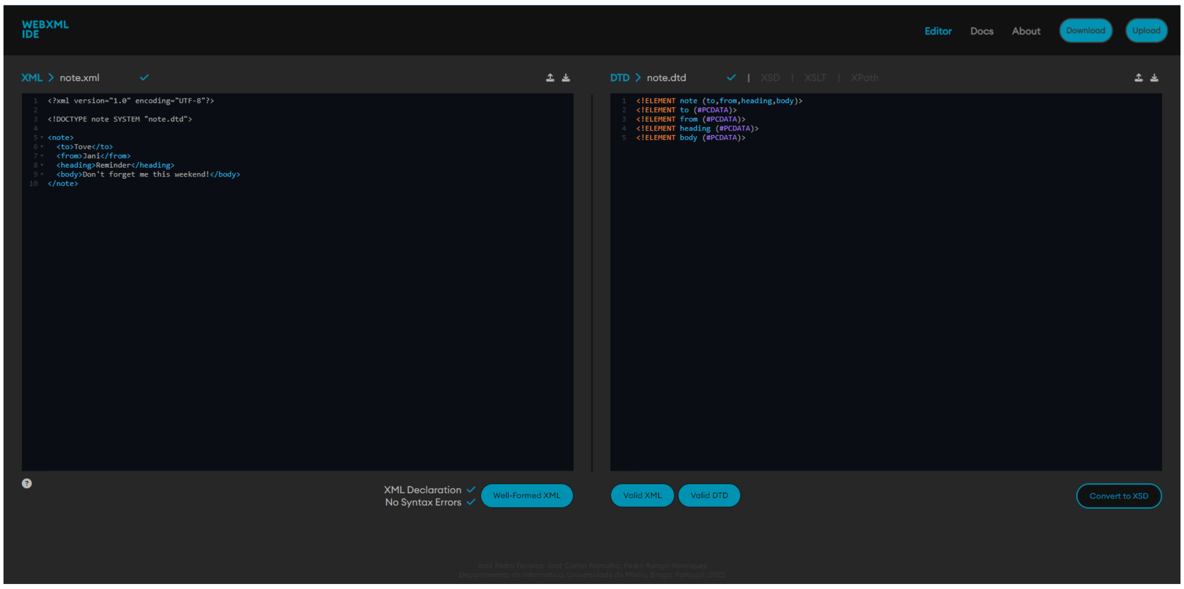
```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Example 6.2: Note.dtd

1 https://www.w3schools.com/xml/xml_dtd.asp

6.1.1 *XML + DTD*

Figure 38 shows the application with an XML Document and a DTD, both well-formed and linked between them. We can see the XML Document is marked as valid against the given DTD.



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

<!DOCTYPE note SYSTEM "note.dtd">

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

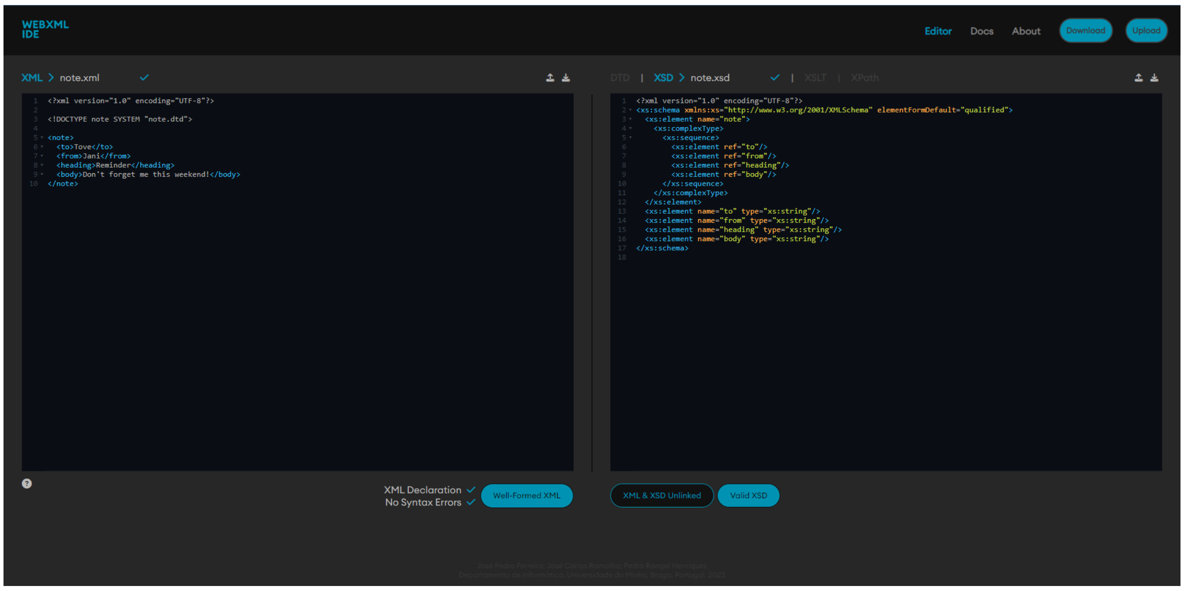
```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

José Pedro Ferreira; José Carlos Ramalho; Pedro Rangel Henriques
Departamento de Informática, Universidade do Minho, Braga, Portugal, 2023

Figure 38: XML + DTD (App Screenshot)

6.1.2 *XML + XSD*

Figure 39 shows the application with an XML Document and a XSD, both well-formed but they're not linked, as the XML Document is linked to the previous DTD. The XSD Code in the screenshot was generated from the DTD Code using the system's converter.



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

<!DOCTYPE note SYSTEM "note.dtd">

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

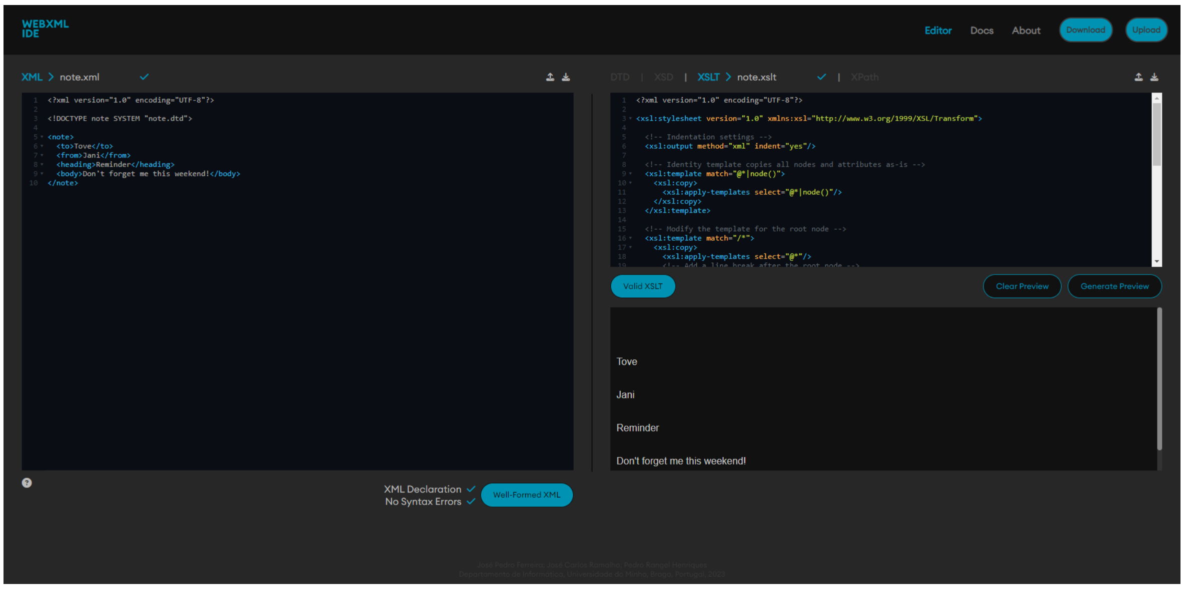
```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="to"/>
        <xs:element ref="from"/>
        <xs:element ref="heading"/>
        <xs:element ref="body"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="to" type="xs:string"/>
  <xs:element name="from" type="xs:string"/>
  <xs:element name="heading" type="xs:string"/>
  <xs:element name="body" type="xs:string"/>
</xs:schema>
```

José Pedro Ferreira; José Carlos Ramalho; Pedro Rangel Henriques
Departamento de Informática, Universidade do Minho, Braga, Portugal, 2023

Figure 39: XML + XSD (App Screenshot)

6.1.3 *XML + XSLT*

Figure 40 shows the application with an XML Document and a XSLT stylesheet, both well-formed. The preview was generated using the system's features.



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

<!DOCTYPE note SYSTEM "note.dtd">

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

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

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

  <!-- Indentation settings -->
  <xsl:output method="xml" indent="yes"/>

  <!-- Identity template copies all nodes and attributes as-is -->
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>

  <!-- Modify the template for the root node -->
  <xsl:template match="/*">
    <xsl:copy>
      <xsl:apply-templates select="@*"/>
      <!-- Add a line break after the root node -->
```

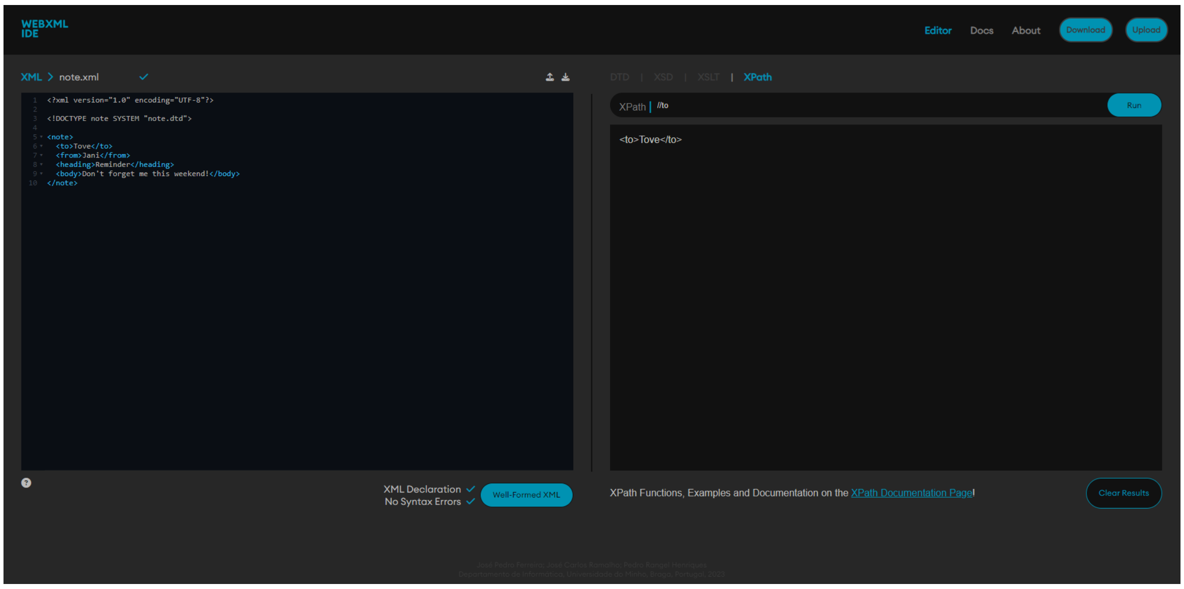
```
Tove

Jani

Reminder

Don't forget me this weekend!
```

José Pedro Ferreira; José Carlos Ramalho; Pedro Rangel Henriques
Departamento de Informática, Universidade do Minho, Braga, Portugal, 2023

Figure 40: XML + XSLT (App Screenshot)

6.1.4 *XML + XPath*

Figure 41 shows the application with a well-formed XML Document and a simple XPath query. The output results from running the XPath expression on the application.



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

<!DOCTYPE note SYSTEM "note.dtd">

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

```
<to>Tove</to>
```

XPath Functions, Examples and Documentation on the XPath Documentation Page!

José Pedro Ferreira; José Carlos Ramalho; Pedro Rangel Henriques
Departamento de Informática, Universidade do Minho, Braga, Portugal, 2023

Figure 41: XML + XPath (App Screenshot)

6.1.5 *Documentation*

Figures 42 and 43 show the documentation page which contains links for tutorials on every component of the IDE, as well as an example project that can be loaded onto the editors and a simple guide on how the project upload works on **WebXMLIDE**.

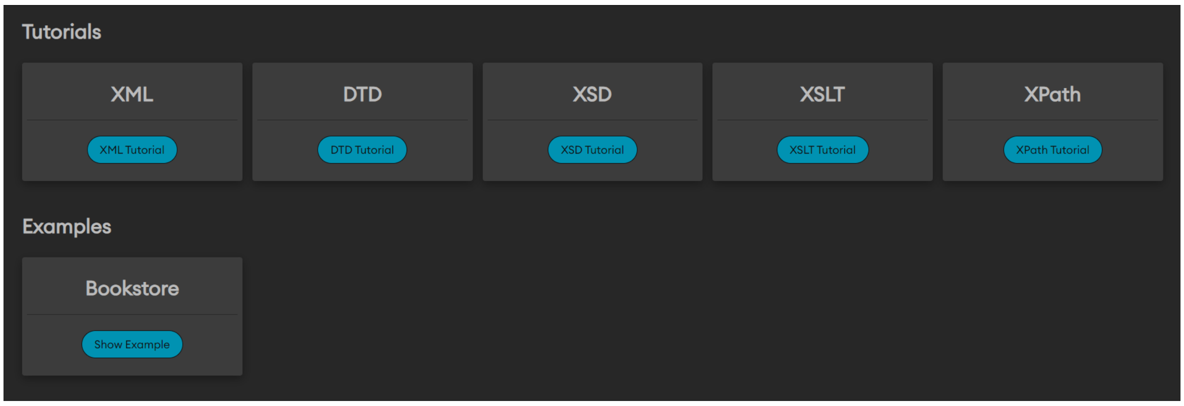
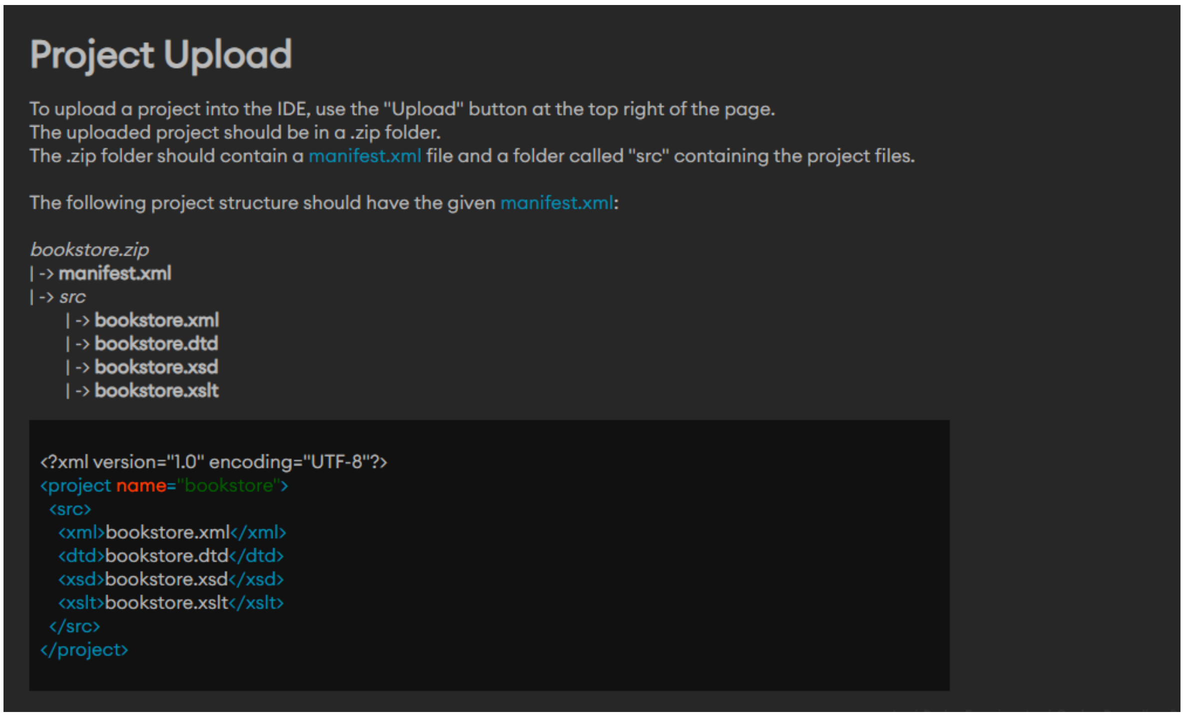


Figure 42: Documentation Page (App Screenshot) - Tutorials and Example Project



Project Upload

To upload a project into the IDE, use the "Upload" button at the top right of the page.
The uploaded project should be in a .zip folder.
The .zip folder should contain a manifest.xml file and a folder called "src" containing the project files.

The following project structure should have the given manifest.xml:

```
bookstore.zip
| -> manifest.xml
| -> src
    | -> bookstore.xml
    | -> bookstore.dtd
    | -> bookstore.xsd
    | -> bookstore.xslt
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="bookstore">
 <src>
  <xml>bookstore.xml</xml>
  <dtd>bookstore.dtd</dtd>
  <xsd>bookstore.xsd</xsd>
  <xslt>bookstore.xslt</xslt>
 </src>
</project>
```

Figure 43: Documentation Page (App Screenshot) - Project Upload Guide

6.2 FEATURE CHECKLIST

Going back to the evaluation criteria defined in Table 5, let's now look at how well the resulting application does when evaluated with the same criteria as the previously analyzed XML IDEs.

To make it more readable, see Table 13.

Requirement	Criteria			
1. XML Code Editor	0 if absent		1 if present	
2. XML Syntax Highlighting, Auto-Completion and Code Folding	0 if none present	1 if 1 is present	2 if 2 are present	3 if all are present
3. XML Document Views: Text, Dev (Text + Annotations) and Tree	0 if none present	1 if 1 is present	2 if 2 are present	3 if all are present
4. XML Well-Formed Check	0 if absent		1 if present	
5. XML Validator according to XSD or DTD	0 if none present	1 if only one present		2 if both present

6. Random XML Documents Generator according to XSD or DTD	0 if none present	1 if only one present		2 if both present
7. XSD Code Editor	0 if absent		1 if present	
8. DTD Code Editor	0 if absent		1 if present	
9. DTD to XSD Converter	0 if absent		1 if present	
10. XSLT Code Editor	0 if absent		1 if present	
11. XSLT Preview	0 if absent		1 if present	
12. XPath Evaluator	0 if absent		1 if present	
13. High Availability (Web-App, Windows, Linux, MacOS)	0 if available only on 1 OS	1 if available only on 2 OS	2 if available on all 3 major OS	3 for WebApp
14. Price	0(Most Expensive) to 4(Free)			
15. Intuitive and simple UI/UX	0 to 10			

Table 13: Evaluation criteria for XML Editors, IDEs and Toolboxes

Table 14 shows the evaluation for the final application:

Requirement	Criteria
1. XML Code Editor	1
2. XML Syntax Highlighting, Auto-Completion and Code Folding	3
3. XML Document Views: Text, Dev (Text + Annotations) and Tree	0
4. XML Well-Formed Check	1
5. XML Validator according to XSD or DTD	2
6. Random XML Documents Generator according to XSD or DTD	0
7. XSD Code Editor	1
8. DTD Code Editor	1
9. DTD to XSD Converter	1
10. XSLT Code Editor	1
11. XSLT Preview	1
12. XPath Evaluator	1

13. High Availability (Web-App, Windows, Linux, MacOS)	3
14. Price	4
15. Intuitive and simple UI/UX	9[2]
Total	29

Table 14: Final Application Evaluation

Comparing the IDE with the highest evaluation during the State of the Art research of this project (given to **Editix**[3]), with the proposed tool and the actual final application for this project, the comparison graph is presented in Figure 44.

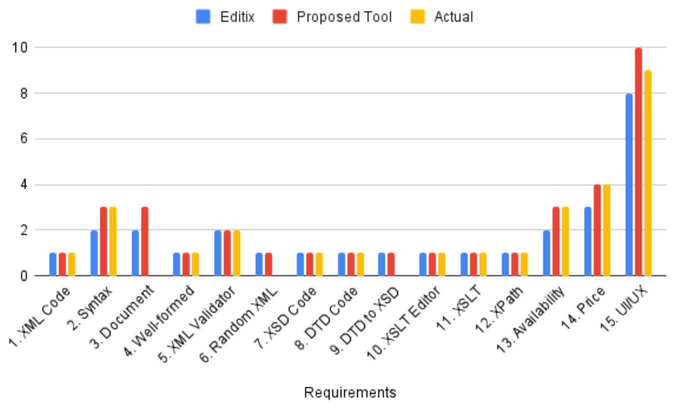


Figure 44: Editix vs Proposed Tool vs Final Product

It is clear some of the features weren't met in the end-product like the different view modes for XML Documents and XML Generation from DTD or XSD. However, the application is still good when put to the test against the best-ranked IDE in **Editix**. Considering the UI/UX criteria is a subjective one, it would be unreasonable to say the final application is better than **Editix** but we can say with certainty that the two IDEs are pretty close to each other.

6.3 SUMMARY

This Chapter provided screenshots of **WebXMLIDE** being used with a real-world example (see Section 6.1), as well as the documentation pages included in the final application, providing users with links to tutorials, project examples and a short guide on how to upload projects to the application (see Subsection 6.1.5).

2 This value is based on user feedback
3 https://www.editix.com/

Section 6.2 presents the results of evaluating **WebXMLIDE** following the criteria defined in Table 13, as well as comparing these results with the best application analyzed in Chapter 3 - **Editix** -, and the initially proposed application.

Having completed the development of **WebXMLIDE**, it was time to test the application with real users. The testing methodology, as well as the results obtained from these tests are presented and analyzed in Chapter 7.

7

TESTS AND RESULTS

This chapter presents the test methodology used to evaluate the success of the project, as well as the results of these tests and the conclusions that can be drawn from these results.

7.1 TESTS

During the development period of the application, a "simplified" version supporting only XML, DTD and XPath was extracted. This "simplified" application was used in practical classes of a Bachelor's Degree curricular unit on Digital Documents and Annotation Languages at Universidade do Minho.

7.1.1 *Methodology*

To understand if the goals of the project were accomplished, it was important to determine who the focus group for this project was and provide them with the opportunity to use the IDE and learn XML with it.

As previously mentioned, the users that used the simplified version of the application were students of a Bachelor's Degree at Universidade do Minho, who were attending a curricular unit on Digital Documents and Annotation Languages. The fact these students came from a (non-technological) linguistics and humanities background, and were learning XML, DTD and XPath, made them the perfect test-users for "WebXMLIDE".

The application was made available on the beginning of April on the URL: `https://webxml.epl.di.uminho.pt/`, and has been used weekly, in these students practical classes, to create XML Documents, define and validate their documents against DTDs, and perform XPath queries on those documents.

After using the application for 7 weeks, these students were asked to answer a survey providing feedback about "WebXMLIDE".

Some experienced users were also given the chance to try the application and answer the survey, in order to see if the application is suitable for all kinds of users.

The survey consisted of 11 mandatory questions with an area for feedback at the very end. These are the questions that were asked in the survey:

1. What was your level of familiarity with programming or annotation languages before using the "WebXMLIDE" tool?
2. Did you find it easy to navigate through the different functionalities (XML, DTD and XPath) in the application?
3. Did you find the tool intuitive and easy to use?
4. How helpful were the code suggestions and auto-complete features when working with XML?
5. Did the application provide enough support to help validate and correct syntax or structure errors in your XML documents?
6. Were you able to perform queries using XPath easily?
7. Did you feel confident using the application to create, edit and manage XML documents?
8. Did you feel confident using the application to create DTDs?
9. Were the error messages and explanations provided by the application clear and helpful in understanding and resolving any issues with your documents?
10. How satisfied are you with the application's graphical interface? (Ease of navigation, design, etc.)
11. Overall, how satisfied are you with the "WebXMLIDE" application in terms of its usability and effectiveness in helping you work with documents?

Optional Any feedback is important to improve the application! If you have any suggestions, please leave them here

These questions were made to understand the Testers background and how it relates to the satisfaction when using the IDE to learn XML and Company.

7.2 RESULTS DISCUSSION

This section presents the responses to the user feedback survey and an analysis of these results.

USER PREVIOUS EXPERIENCE The user feedback survey received 47 responses from users of different experience-levels with programming and annotation languages (non-experienced users represent the majority of responses, as shown in Figure 45).

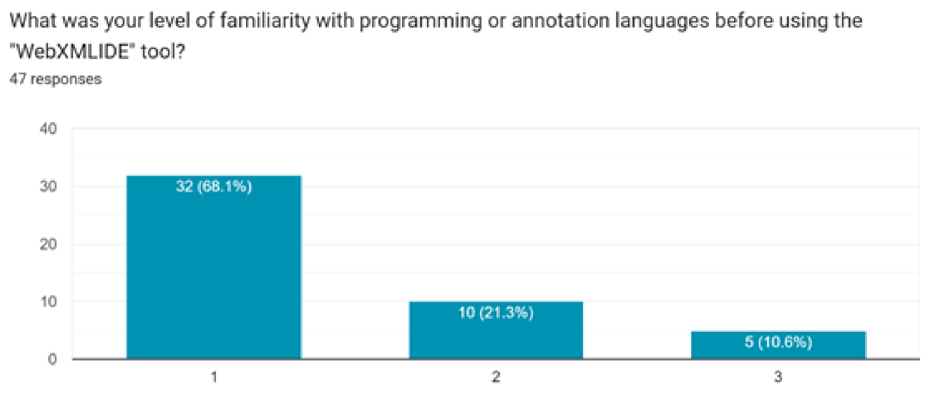


Figure 45: Forms Question 1 Responses - User Previous Experience

APPLICATION NAVIGATION AND EASE-OF-USE One very important aspect of the UI design for this project was to make it easy for beginner-level users to navigate around the website and the different functionalities, which was a problem in many of the IDEs analyzed in Chapter 2. This was achieved successfully, as proven by the responses on application navigation and ease-of-use, as can bee seen in the graphics of Figures 46 and 47.

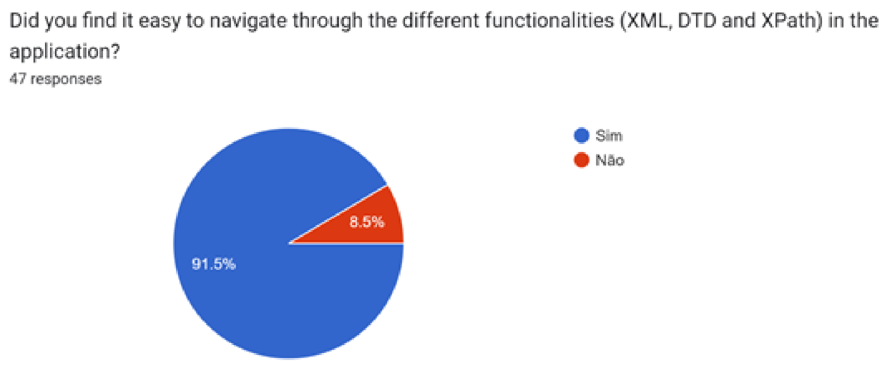


Figure 46: Forms Question 2 Responses - Navigating the App

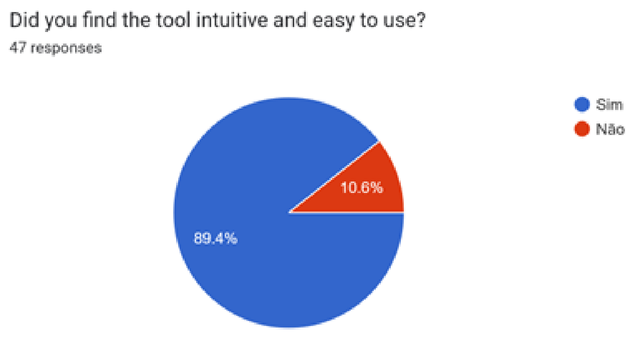


Figure 47: Forms Question 3 Responses - Application Ease-of-use

CODING-SUPPORT FEATURES Another aspect that was defined as key to ensure good support when writing XML Documents in a Code Editor was the Coding-Support Features like Code Suggestions and Auto-Complete Features. The graphics in Figure 48 illustrates the results obtained.

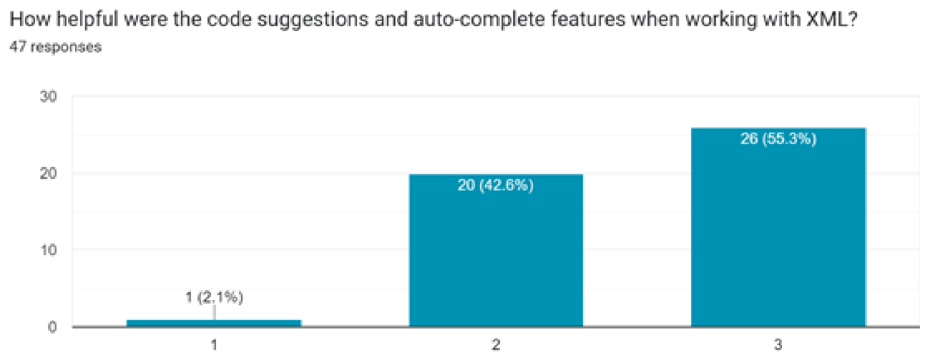


Figure 48: Forms Question 4 Responses - Coding-Support Features

SUPPORT IN FINDING AND FIXING ERRORS IN XML DOCUMENTS Displaying any syntax or structural errors in XML Documents is one of the most important aspects when it comes to helping beginner-level users feel confident in creating XML Documents.

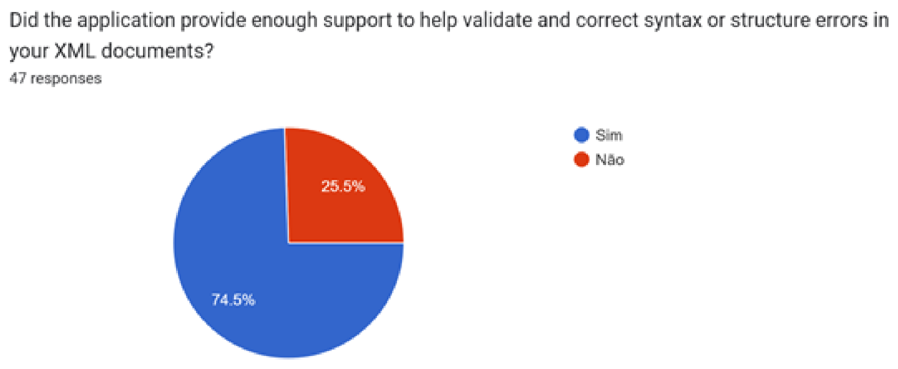


Figure 49: Forms Question 5 Responses - Support in Finding and Fixing Errors in XML Documents

As can be seen in the graphics of Figure 49, there were 25.5% responses stating the given support on identifying errors on XML Documents was not good enough and the responses to the optional user feedback question at the end of the survey explained why this happened: The errors are being displayed in a way that the UI is not ready to handle a massive list of errors. Some errors won't be visible because the size of the error message box. This is a front-end error that can be easily fixed. Another explanation given to these results is based on the error messages provided. As these messages come directly from the JavaScript libraries that were used to parse and validate XML Documents, DTDs, XML Schemas or XSLT Stylesheets, there was little-to-no control over these messages, which is something that could be improved in the future. The responses to a similar question (but applied to all the different components) show this was one of the least successful aspects of the "simplified" version of **WebXMLIDE** that was put to test, as can be confirmed in the graphic of Figure 50.

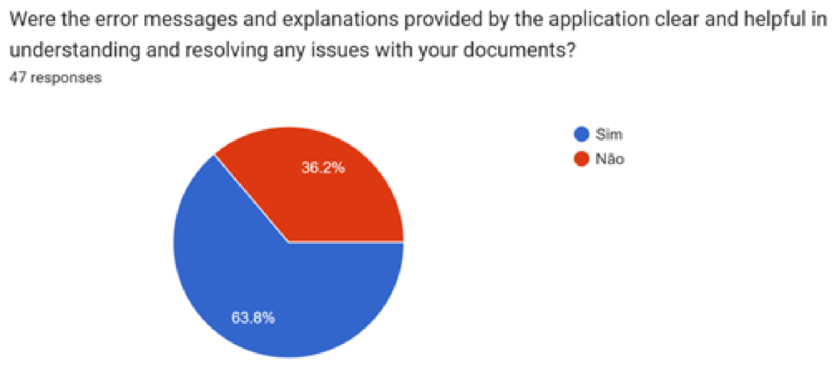


Figure 50: Forms Question 9 Responses - Support in Finding and Fixing Errors

USER CONFIDENCE IN BUILDING XML DOCUMENTS USING WEBXMLIDE The application offers Coding-Support Features and Support in Finding and Fixing Errors in XML Documents to the users with the intent of making them feel confident in using **WebXMLIDE** to create, edit and manage XML Documents. This was successfully achieved as shown in Figure 51.

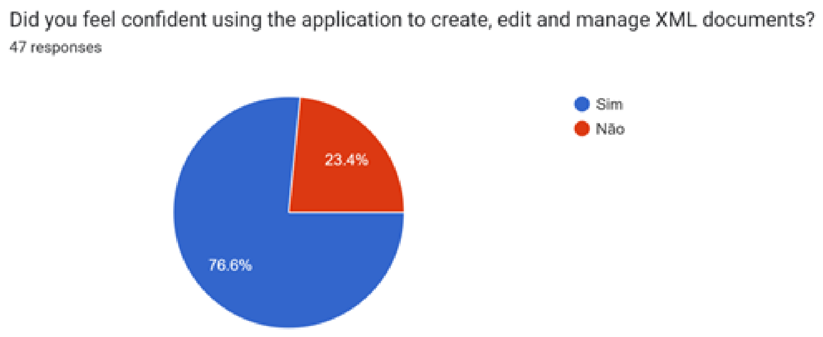


Figure 51: Forms Question 7 Responses - User Confidence in Building XML Documents using WebXMLIDE

USER CONFIDENCE IN BUILDING DTDS USING WEBXMLIDE Helping users create, edit and manage DTDs was also a very important part of the real-world testing phase of this project. DTDs were the most different component (XML, XSD and XSLT are all XML-Syntax-Based), so it was important to make sure users were confident in using the application to build this type of specification file. The majority of test users felt confident using the application to build DTDs, as shown in the graphic of Figure 52.

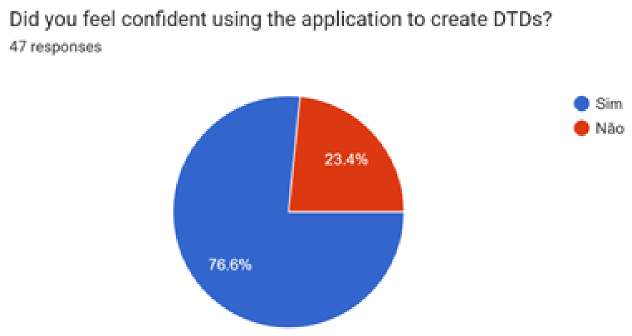


Figure 52: Forms Question 8 Responses - User Confidence in Building DTDs using WebXMLIDE

USER ABILITY TO PERFORM XPATH QUERIES The simplified version of the application allowed users to run XPath Queries on XML Documents. Allowing users to test XPath expressions on their documents is very important as it allows to show that the user can fetch specific elements, text, etc. from a document, which is a very useful and important part of the XML work environment. The answers described in the graphic of Figure 53 prove that the XPath component was succesfully implemented.

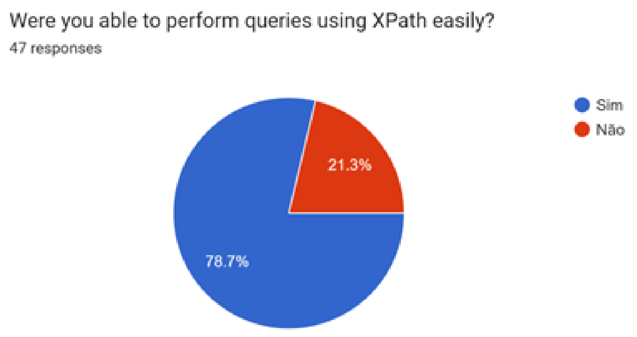


Figure 53: Forms Question 6 Responses - User Ability to Perform XPath Queries

UI SATISFACTION Most of the State of the Art IDEs had problems with the presented UI. It was important to make sure this application presented the users with an User Interface that would be easy to navigate and use, while at the same time featuring a modern and simplistic design.

The responses depicted in the graphic of Figure 54 describe the user satisfaction on the application's graphical interface.

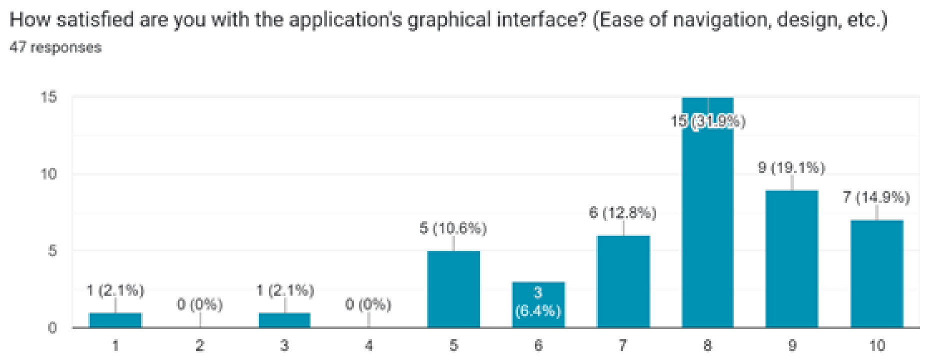


Figure 54: Forms Question 10 Responses - UI Satisfaction

OVERALL SATISFACTION The last question on the survey served the purpose of understanding the users' overall satisfaction with the provided application. The results described in graphics format in Figure 55 show a very good number of users who are satisfied with the application.

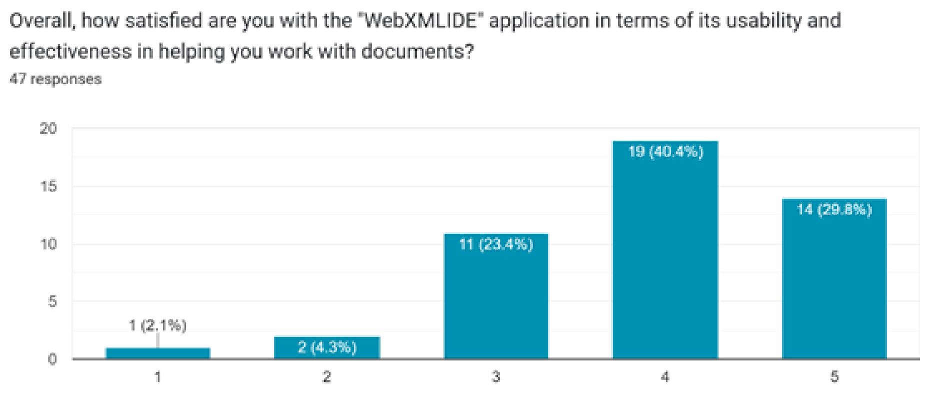


Figure 55: Forms Question 11 Responses - Overall Satisfaction

analysis summary From the responses received in the survey, it is easy to notice that the simplified version of the **WebXMLIDE** application had some flaws, specially in displaying the error messages, and the error messages themselves. However, the overall satisfaction results show that the application served the needs of the majority of users.

8 CONCLUSION

This last chapter presents the work plan and timeline, as well as a summary of this thesis, a reflection on the highlights and contributions achieved in this project, and the future work that could be done to improve the application.

8.1 WORK PLAN

This section will present the work plan defined at the beginning of this project and then present the work timeline that happened while developing this project.

This Master's thesis was estimated to be completed in one academic year. The development of this project was scheduled according to the following phases:

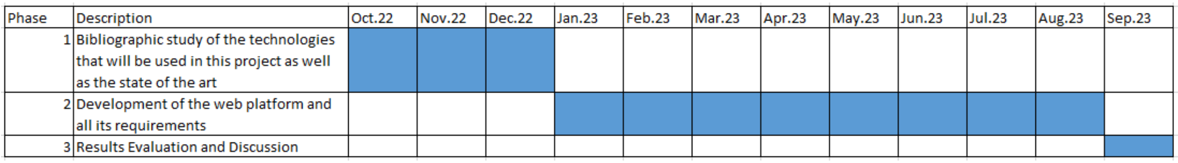
1ST TO 3RD MONTH: The first three months are focused on the bibliographic study of the technologies that will be used in this project as well as the state of the art.

4TH TO 11TH MONTHS: Development of the web platform fulfilling the proposed requirements.

12TH MONTH: Results Evaluation and Discussion.

The development of this thesis report was supposed to occur simultaneously to the development of the each phase previously mentioned. The final stage of this project was set to be almost exclusively focused on the conclusions over the work done and reviewing the thesis report.

The Gant Chart in Figure 56 shows the previously explained work plan.



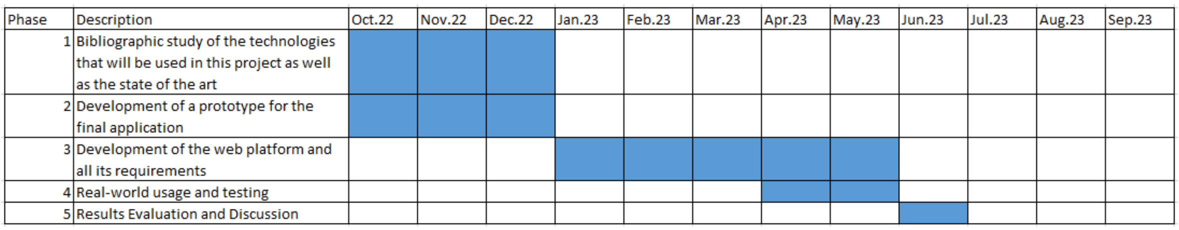
Phase	Description	Oct.22	Nov.22	Dec.22	Jan.23	Feb.23	Mar.23	Apr.23	May.23	Jun.23	Jul.23	Aug.23	Sep.23
1	Bibliographic study of the technologies that will be used in this project as well as the state of the art	🟦	🟦	🟦									
2	Development of the web platform and all its requirements				🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	
3	Results Evaluation and Discussion												🟦

Figure 56: Planned Schedule of the project

The time spent to develop this project was actually shorter than planned and included two more phases: one where a prototype was built, to serve as a playground to test and play with the different technologies that would later be used on the final application; and another, where a simplified version of the application (XML with DTD and XPath only) was put to use in a real world context so that it could be tested and improved. The complete version was made available to test as well.

The writing of this report always went hand-on-hand with the development of the project.

The Gant Chart in Figure 57 shows the actual work timeline.



Phase	Description	Oct.22	Nov.22	Dec.22	Jan.23	Feb.23	Mar.23	Apr.23	May.23	Jun.23	Jul.23	Aug.23	Sep.23
1	Bibliographic study of the technologies that will be used in this project as well as the state of the art	🟦	🟦	🟦									
2	Development of a prototype for the final application	🟦	🟦	🟦									
3	Development of the web platform and all its requirements				🟦	🟦	🟦	🟦	🟦				
4	Real-world usage and testing							🟦	🟦				
5	Results Evaluation and Discussion									🟦			

Figure 57: Actual Work Timeline of the project

8.2 REPORT SUMMARY

So far, this document has given the context and motivation for the development of the proposed tool, as well as an explanation on well-formed XML Documents, XML Document validation against DTD or XML Schemas and some components of XSL (XSLT and XPath).

After introducing the project for this Master's thesis, this document showed an extensive research on the available XML IDEs, XML Editors and some XML Toolboxes, analyzing how well each product satisfies the requirements established for this project and comparing the best option available (according to the criteria defined in Table 5 and the results of evaluating each tool according to the criteria in Table 6) to the proposed tool. The results from this research reinforced the research hypothesis (Chapter 1.3) by showing that, despite the existence of some very complete solutions for inexperienced XML users to work on XML Development, these are often too complicated to use, unavailable for all users, not appropriate for beginners, very expensive or simply lack too many features that are key when learning the basics of XML Development. The absence of a beginner-oriented, free-to-use, highly available (as a web application) and free solution makes the case for why the proposed tool is not only viable but necessary.

Then, Chapter 4 explained the proposed approach for the development of the desired Web XML IDE, presented the system's use cases, provided an explanation on how the system will deal with input files and the processing components, and presented the mockups for the proposed application.

The fifth chapter presented the development of this project in two different moments (planning and implementing stages) and the technologies that were used in the process. It offered a more in-depth view of the system's requirements and architecture as well as a brief and simple explanation on how the many different features were implemented in the application.

The sixth chapter has presented the end-product, named **WebXMLIDE**, using screenshots and taking the final app to the test on the criteria defined and used in Chapter 2, in order to compare the feature list of **WebXMLIDE** against what was proposed with the project and what was already available in the market when it came to XML IDEs.

The seventh chapter introduced the testing methodology and the results from those tests, evaluating user feedback and determining if the goal of the project was accomplished or not.

This last chapter presents the work plan defined at the beginning of the project and the actual work plan, a small summary of this document, a reflection on the highlights of this project and some ideas for future work that would benefit this project.

8.3 Highlights and Contributions

The developed application met most of the requirements and objectives initially proposed, which in itself is a highlight, but one of the biggest achievement was to have the application available to use in the real-word for users to learn XML while using it, providing the opportunity for testing the software with the desired focus group in a real context. The testing phase turned out to be a success and a very important stage of the application's development.

The biggest achievement of this project is the fact that there is a final version of the application running on the internet, accessible to everyone and free, offering a high-quality and beginner-friendly platform for working with XML & Company.

8.4 Future Work

While developing the application, it was already possible to imagine some tools, libraries or features that would make the development process or the application itself better.

One thing that would made the development of this project a lot easier is a JavaScript library ready to deal with all different parts of working with XML. Finding a library that supported DTDs was a difficult challenge that was overcome but the same library didn't support XPath expressions, or at least, not well enough. This is the reason why there are 3 different JavaScript libraries and 1 command line component to deal with XML in this project: *node-libxml* was used for parsing XML, DTD and XSD Files as well as test the validity of XML Documents against a given DTD or Schema; *Libxmljs2* was used to run XPath expressions on the XML Documents (*node-libxml* provided a very limited XPath feature, hence the need to use a different library); *xmldom* was used to parse XML Documents and create DOM Trees from the input or source documents; last but not least, *Trang* was used to convert DTDs into XML Schemas. **A JavaScript library to support all of these features** would be a massive help to developers working with XML & Company on JavaScript projects.

There was also a lack of good options to generate XML code from a given DTD or XML Schema. There were some available, but all of them provided a different set of limitations and none would be able to create viable and valid XML Documents based on DTDs or XSDs. One thing that came up was the possibility to build, not only a generator capable of creating XML Documents from **any** DTD or Schema, but also a context-sensitive one. This idea comes from the increased use of AI to support developers in programming. An AI model that would be able to generate well-formed XML Documents that would be valid against the input DTD or Schema, but also sensitive to the tags and schema definition, when it came to filling the text elements or attributes. This would lead to realistic, well-formed and valid XML Document generation.

Besides the lack of libraries and tools described previously, there are also some improvements that could be made to the application, slightly improving User Experience and functionality, which are two key factors given the goals for this project. One of these things is **an interactive and graphical way to create Schemas**. Some editors already allow for the user to create XML Schemas using graphical editors where the user drags and drops different elements to create the desired rules for the schema. Schemas are heavy to write by hand, even with auto-completion and syntax highlighting. Adding this feature would improve on the ease-of-use and improve the application's support to learning XSD.

Another improvement that could be made to this project would be to use the *hint-options* configuration of **CodeMirror** with real-time DTD or XSD conversion to simple JavaScript Schemas, using the DTD or XSD defined in the application to **provide the user with code suggestions that match the entire DTD or XSD**, instead of providing them only taking into consideration the previously opened element tag.

BIBLIOGRAPHY

James Clark. Trang. https://relaxng.org/jclark/trang.html, 2008. Accessed: 2023-03-30.

Bigyan Ghimire. 15 Best XML Editors for Productive Development. https://geekflare.com/best-xml-editors/, November 2022. Accessed: 2022-11-26.

Software Testing Help. 14 Best XML Editors In 2022. https://www.softwaretestinghelp.com/best-xml-editors/, October 2022. Accessed: 2022-11-29.

ArborText Inc. W3C XML Specification DTD ("XMLspec"). https://www.w3.org/XML/1998/06/xmlspec-report-v20.html, June 1998. Accessed: 2022-09-14.

Michael Dyck Jonathan Robie and Josh Spiegel. XML Path Language (XPath) 3.1. https://www.w3.org/TR/2017/REC-xpath-31-20170321/, March 2021a. Accessed: 2022-11-26.

Michael Dyck Jonathan Robie and Josh Spiegel. XQuery 3.1: An XML Query Language. https://www.w3.org/TR/xquery-31/, March 2021b. Accessed: 2022-11-26.

Michael Kay. XSL Transformations (XSLT) Version 2.0 (Second Edition). https://www.w3.org/TR/2021/REC-xslt20-20210330/, March 2021. Accessed: 2022-11-26.

James Kiarie. The Most Used Operating Systems in the World. https://www.tecmint.com/most-used-operating-systems-world/, November 2022. Accessed: 2022-12-05.

James Porter. Vue Components. https://blog.scottlogic.com/2020/09/22/vue-components.html, September 2022. Accessed: 2023-03-15.

Liam Quin. Extensible Markup Language (XML). https://www.w3.org/XML, October 2016. Accessed: 2022-11-3.

Liam Quin. The Extensible Stylesheet Language Family (XSL). https://www.w3.org/Style/XSL, September 2017. Accessed: 2022-11-2.

José Carlos Ramalho and Pedro Rangel Henriques. *XML & XSL: da teoria à prática*. Série Tecnologias de Informação ISBN-972-722-347-8. Editora FCA, 1st ed. edition, Oct 2002.

C. M. Sperberg-McQueen and Henry Thompson. W3C XML Schema. https://www.w3.org/XML/Schema, April 2000. Accessed: 2022-09-14.