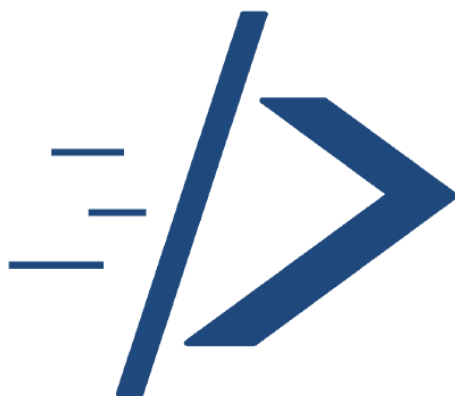




INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores



Automatic generation of a Java API based on XML Schema

LUÍS CARLOS DA SILVA DUARTE

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: [Grau e Nome do presidente do júri]

Vogais: [Grau e Nome do primeiro vogal]

[Grau e Nome do segundo vogal]

[Grau e Nome do terceiro vogal]

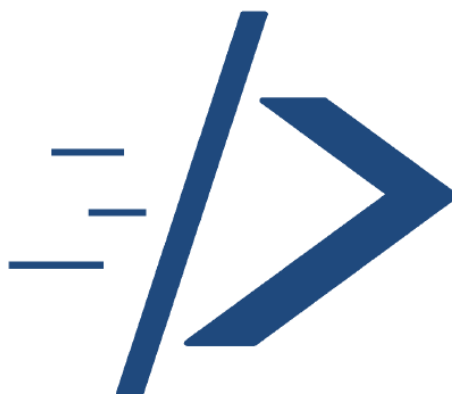
[Grau e Nome do quarto vogal]

Junho, 2018



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores



Automatic generation of a Java API based on XML Schema

LUÍS CARLOS DA SILVA DUARTE

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: [Grau e Nome do presidente do júri]

Vogais: [Grau e Nome do primeiro vogal]
[Grau e Nome do segundo vogal]
[Grau e Nome do terceiro vogal]
[Grau e Nome do quarto vogal]

Junho, 2018

Aos meus pais.

Acknowledgments

Ao meu orientador, por todo o apoio que me deu ao longo da realização desta dissertação. A todos os meus amigos que me acompanharam, ajudaram e animaram nesta jornada. E em particular, um grande agradecimento aos meus pais, sem eles nada disto seria possível.

Acronyms and Abbreviations

The list of acronyms and abbreviations are as follow.

API <i>Application Programming Interface</i>	xi
DOM <i>Document Object Model</i>	16
HTML <i>HyperText Markup Language</i>	xi
IDE <i>Integrated Development Environment</i>	36
JMH <i>Java Microbenchmark Harness</i>	43
JSP <i>JavaServer Pages</i>	12
POM <i>Project Object Model</i>	35
SAX <i>Simple Application Programming Interface for eXtensive Markup Language</i> ..	16
URL <i>Uniform Resource Locator</i>	47
XHTML <i>eXtensive HyperText Markup Language</i>	16
XML <i>eXtensive Markup Language</i>	6
XSD <i>eXtensive Markup Language Schema Definition</i>	xi

Abstract

The use of markup languages is recurrent in the world of technology today, with *HyperText Markup Language* (HTML) being the most prominent one due to its use in the Web. The need of tools that can automatically generate well formed documents with good performance is clear. Currently in order to tackle this problem the most used solution are template engines which base their solution on the usage of an external file, which doesn't ensure well formed documents and introduces the overhead of loading the template files to memory which degrades the overall performance.

Our objective is to create the required tools to generate fluent *Application Programming Interface* (API)s based on a language definition file, *eXtensive Markup Language Schema Definition* (XSD), while enforcing the restrictions of the given language. The generation of the APIs should be automated in order to avoid human error and expedite the coding process. By automating the API generation we also create a uniform approach to these languages.

To achieve our objectives we will use the Java language to extract the data from the language definition file. Based on the information provided by the language definition file we can then generate the adequate bytecodes to reflect the language definition to the Java language. To implement the language restrictions in Java we will always prioritize compile time validations, only performing run time validations of information that isn't available when the API is generated.

By comparing the developed solution to some existing solutions, including ten template engines and one other solution similar to the one we are proposing, we obtained very favorable results with the suggested solution being the best performance-wise in all the tests we performed. These results are important, specially considering that apart from being a more efficient solution it also introduces

validations of the language usage based on its syntax definition.

Keywords: XML, XSD, Automatic Code Generation, fluent API.

Resumo

Actualmente a utilização de linguagens de markup é recorrente no mundo da tecnologia, sendo o HTML a linguagem mais utilizada graças à sua utilização no mundo da Web. Tendo isso em conta é necessário que existam ferramentas capazes de escrever documentos bem formados de forma eficaz. Actualmente essa tarefa é realizada por template engines, tendo como base ficheiros externos com templates de resposta, o que não garante que estes sejam bem formados e acrescenta o overhead do carregamento do ficheiro para memória.

O nosso objectivo é criar as ferramentas necessárias para gerar APIs fluentes tendo em conta a sua definição sintática, expressa em XSD, garantindo que as restrições dessa mesma linguagem são verificadas. A geração de APIs deve ser automatizada de modo a evitar erro humano e tornar a geração de código mais rápida. Automatizando a geração de APIs cria-se também uma abordagem uniforme às diferentes linguagens utilizadas.

Para alcançar os nossos objectivos vai ser utilizada a linguagem Java para extrair informação sintática da linguagem do seu ficheiro de definição. Tendo essa informação em conta vão ser gerados bytecodes para refletir a definição da linguagem para a linguagem Java. Para implementar as restrições em Java é sempre priorizada a validação de restrições em tempo de compilação, apenas validando em tempo de execução informação que não existe aquando da geração da API.

Comparando a solução desenvolvida com soluções semelhantes, incluindo dez template engines e outra solução semelhante à que é apresentada, obtemos resultados favoráveis, verificando que a solução sugerida é a mais eficiente em todos os testes feitos. Estes resultados são importantes, especialmente considerando que apesar de ser a solução mais eficiente introduz também a verificação das restrições da linguagem utilizada tendo em conta a sua definição sintática.

Palavras-chave: XML, XSD, Geração Automática de Código, API fluente.

Contents

List of Figures	xvii
List of Listings	xix
1 Introduction	1
1.1 Scope	1
1.2 Template Engines Handicaps	4
1.3 Thesis statement	6
1.4 Document Organization	6
2 Problem Statement	7
2.1 Motivation	7
2.2 Use case	9
3 State of Art	11
3.1 XSD Language	11
3.2 Template Engines	12
3.2.1 Xmllet Similarities	12
3.2.2 J2html	13
3.2.3 Apache Velocity	13

4	Solution	15
4.1	XsdParser	15
4.1.1	Parsing Strategy	16
4.1.2	Reference solving	20
4.2	XsdAsm	22
4.2.1	Supporting Infrastructure	22
4.2.2	Code Generation Strategy	24
4.2.2.1	Type Arguments	26
4.2.3	Restriction Validation	29
4.2.3.1	Enumerations	32
4.2.4	Element Binding	33
4.3	Client	35
4.3.1	HtmlApi	35
4.3.1.1	Using the HtmlApi	38
5	Deployment	41
5.1	Github Organization	41
5.2	Maven	41
5.3	Sonarcloud	42
5.4	Testing metrics	42
5.4.1	Spring benchmark	45
6	Conclusion	51
6.1	Future work	52
6.1.1	Early Results	53

List of Figures

1.1	Template Engine Process	2
2.1	Error validation in compile time	9
4.1	API - Supporting Infrastructure	23
5.1	XsdParser Github badges	42
5.2	Benchmark: Generating an Html Table with 10 Elements	44
5.3	Benchmark: Generating an Html Table with 100 Elements	44
5.4	Benchmark: Generating an Html Table with 1.000 Elements	44
5.5	Benchmark: Generating an Html Table with 10.000 Elements	45
5.6	Benchmark: Spring Benchmark with 10.000 requests	48
5.7	Benchmark: Spring Benchmark with 30.000 requests issued with 10 threads	48
5.8	Benchmark: Spring Benchmark with 30.000 requests issued with 25 threads	49
5.9	Benchmark: Spring Benchmark with 100.000 requests	49

List of Listings

1.1	Dynamic Hello	2
1.2	Xmllet Dynamic Hello	4
2.1	Failed HTML rule validation	8
2.2	Lack of rule validation	8
4.1	XsdAnnotation class (Simplified)	16
4.2	DOM Document Parsing	16
4.3	XsdParser Node Parsing Process	17
4.4	XsdSchema Information Extraction (Simplified)	17
4.5	XsdParseSkeleton Parsing Children From a Node	18
4.6	Parsing Concrete Example	19
4.7	Reference Solving Example	21
4.8	Code Generation XSD Example	24
4.9	Html Element Class	25
4.10	Body Element Class	25
4.11	Head Element Class	25
4.12	Manifest Attribute Class	25
4.13	CommonAttributeGroup Interface	26
4.14	Explicit Example of Type Arguments	27
4.15	Simpler Example of Type Arguments	27
4.16	Type Arguments - AbstractElement class	28

4.17	Type Arguments - AbstractElement class	28
4.18	Type Arguments - AbstractElement class	29
4.19	Restrictions Example	29
4.20	Attribute Class Example	30
4.21	Attribute Static Constructor Restrictions	30
4.22	BaseAttribute Rule Validation Restrictions	31
4.23	Restriction Validator Class (Simplified)	32
4.24	Enumeration XSD Definition	32
4.25	Enumeration Class	33
4.26	Attribute Receiving An Enumeration	33
4.27	Binder Usage Example	33
4.28	Visitor with binding support	34
4.29	API creation	35
4.30	Maven API compile classes plugin	36
4.31	Maven API creation batch file (create_class_binaries.bat)	36
4.32	Maven API decompile classes plugin	37
4.33	Maven API decompile batch file (decompile_class_binaries.bat)	37
4.34	Custom Visitor	38
4.35	HtmlApi Element Tree	39
4.36	HtmlApi Visitor Result	40
5.1	Test HTML	43
5.2	Spring Benchmark Mustache Template	46
5.3	Spring Benchmark Presentation Object	47
6.1	Future Solution Usage 1	52
6.2	Future Solution Html Class (Simplified)	52
6.3	Future Solution Body Class (Simplified)	53



Introduction

The research work that I describe in this dissertation is concerned with the implementation of a Java framework, named `xmlet`, which allows the automatic generation of a fluent API based on a XSD file.

1.1 Scope

The work that is presented in this dissertation will be about Java applications that use *dynamic views*. A *dynamic view* is a concept that can be described as view that contains two distinct parts:

- Static part - Represented by all the information of the view that doesn't depend on any kind of external input.
- Dynamic part - Represented by placeholders that should be replaced at run-time with information received from a source of input.

A simple example of a *dynamic view* can be an HTML page that greets a given user as shown in Listing 1.1.

```
1 <html>
2   <body>
3     <div>
4       <b>
5         Hello
6       </b>
7       <i>
8         {{userName}}
9       </i>
10    </div>
11  </body>
12 </html>
```

Listing 1.1: Dynamic Hello

In this example we can observe the distinct parts: 1) the dynamic part is represented by `{{userName}}` and 2) the static part is represented by all the remaining information of the example. To generate a complete view this dynamic view needs to receive information at runtime to replace the dynamic aspect of the view, in the previous example, Listing 1.1, the view needs to receive a value for the variable named `{{userName}}`. The example presented in Listing 1.1 is defined in the Mustache¹ idiom, which is a template engine with implementations for the most used programming environments, including Java.

The most common method to manipulate *dynamic views* are *template engines*. Template engines are responsible for performing the combination between the *dynamic view*, also named *template*, and a data model, which contains all the information required to generate a complete document as shown in Figure 1.1.

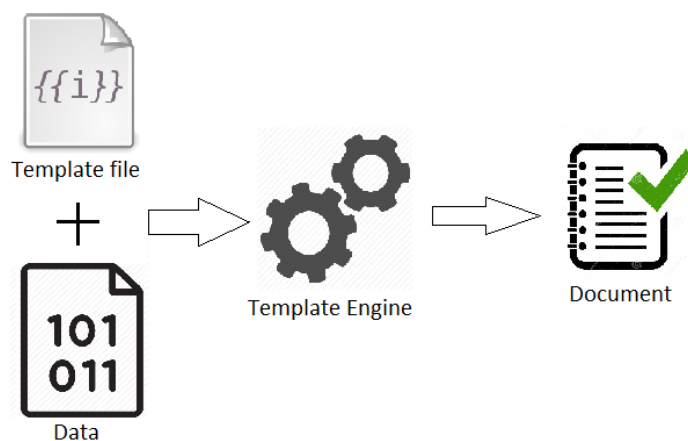


Figure 1.1: Template Engine Process

¹Mustache [GitHub](#)

Since the Web appearance to this day there is a wide consensus around the use of template engines to generate dynamic HTML document generation. The consensus is such that there isn't a real alternative to the usage of template engines for dynamic generation of documents. The template engine scope is also wide, even that they are mostly associated with Web and its associated technologies they are also widely used to generate other types of documents such as emails and reports. Although there is a wide consensus in the usage of this kind of solution it still contains some handicaps, we will list the four main handicaps of template engines:

- Language Compilation - There is no compilation of the language used in the templates nor the dynamic information. This can result in documents that don't follow the language rules.
- Performance - This aspect can be divided in two, one regarding the text files that are used as templates which have to be loaded and therefore slow the overall performance of the application and the heavy usage of string operations which are inherently slow.
- Flexibility - The syntax provided by the template engines is sometimes very limited which limits the operations that can be performed in the template files, often to if/else operations and a for each to loop data.
- Complexity - It introduces more languages to the programmer, for example a Java application using the Mustache template engine to generate HTML forces the programmer to use three distinct languages, Java, the Mustache syntax in the template file and the HTML language.

To suppress all the handicaps presented above we propose a new solution, `xmlet`, which allows the automatic creation of a strongly typed and fluent API for a specific domain language defined with its syntax defined in a XSD file, such as HTML. How will the `xmlet` solution address the handicaps of the template engines?

- Language Compilation - The generated API will guarantee the implementation of the language restrictions defined in the XSD file by reflecting those restrictions to the Java language.
- Performance - The text files to contain templates are replaced by Java functions that represent templates, removing the need to load an additional file.

- Flexibility - The syntax to perform operations on templates is changed to the Java syntax which is much more flexible than any template engine syntax.
- Complexity - It removes the use of three distinct languages, the programmer only needs to program in Java.

To understand how the generated API will work we will present a little example, Listing 1.2, that shows how the previous example in the Mustache idiom (Listing 1.1) will be recreated with the `xmlet` solution. The specific details on how the code presented in this example works will be provided in the Chapter 4.

```
1 Html<Html> html = new Html<>();  
2  
3 String userName = "Luis";  
4  
5 html.  
6     body()  
7         .div()  
8             .b()  
9                 .text("Hello") .°()  
10                .i()  
11                .text(userName);
```

Listing 1.2: Xmlet Dynamic Hello

1.2 Template Engines Handicaps

Currently there are dozens of different template engine idioms implemented for a vast diversity of programming environments. Despite the vast number of different solutions they all share the same approach depicted in Figure 1.1. This means that they all share the handicaps enumerated previously. These handicaps tend to become even worse when the complexity of the template escalates. How do the handicaps behave when the complexity escalates then?

- Language Compilation - With more complex templates the language violation tend to be more frequent.
- Performance - Often to ease the complexity of the template the file is divided in multiple files (which facilitates re usability), this increases the number of text files that the template engine has to load. Regarding string operations,

the bigger the template the slower the string operations will become and most likely the number of string operations will also escalate with the number of placeholders.

- Flexibility - With poor flexibility regarding operations in the template engine syntax the complexity will generate complex templates which will become hard for programmers to understand and write.
- Complexity - The complexity of having three languages escalates with the complexity of the template files, since the programmer has to be focused not only on the inherent complexity of the template but on the complexity of all the different syntaxs that are being used.

The solutions that existed prior to this dissertation were already aiming to solve some of those handicaps. We had two different solutions, J2html², which is a solution that removes the necessity of text files to define the templates and using some Java instructions to replace the syntax provided by template engines. J2html however did not guaranteed that the language rules were being followed while creating the documents and is only designed to work for HTML. The second solution, HtmlFlow³ removed the necessity of text files to define the templates and also used the Java syntax to manipulate the templates. HtmlFlow had already implemented some restrictions of the HTML language manually but only supported a few core HTML elements, since recreating all the HTML elements and their restrictions manually is very time consuming. Both of theses solutions had also problems regarding maintainability, if any change was needed it had to be performed manually.

While both these solutions were a step in the right direction some aspects became clear, we needed to create a process that could automatically create a fluent API based on the set of rules of a given language. The generated APIs should take advantage of the Java language compiler to enforce the language restrictions and use its syntax to generate templates without having the need to create templates in text files.

²J2html

³HtmlFlow

1.3 Thesis statement

This dissertation thesis is that it is possible to reduce the time spent by programmers by creating a process that automatizes the creation of fluent APIs based on a set of rules present in a XSD file. The process encompasses three distinct aspects:

- XsdParser - Which parses the XSD file in order to extract information needed to generate the API.
- XsdAsm - Which uses XsdParser to extract the information needed to generate the API and uses it to generate an API.
- HtmlApi - A concrete API generated by XsdAsm using the HTML5 XSD file.

The use case used in this dissertation will be the HTML language but the process is designed to support any domain language that has its definition in the XSD syntax. This means that any *eXtensive Markup Language* (XML) language should be supported as long as it has its set of rules properly defined in a XSD file. To show that this solution is viable with other XSD files we used another XSD file that detailed the rules of the XML syntax used to generated Android⁴ visual layouts.

1.4 Document Organization

This document will be separated in six distinct chapters. The first chapter, this one, introduces the concept that will be explored in this dissertation. The second chapter introduces the motivation for this dissertation. The third chapter presents existent technology that is relevant to this solution. The fourth chapter explains in detail the different components of the suggested solution. The fifth chapter approaches the deployment, testing and compares the **xmlet!** (**xmlet!**) solution to other existing solutions. The sixth and last chapter of this document contains some final remarks and description of future work.

⁴Android

Problem Statement

2.1 Motivation

Text has evolved with the advance of technology resulting in the creation of *markup languages* [?]. Markup languages work by adding annotations to text, the annotations being also known as tags, that allow to add additional information to the text. Each markup language has its own tags and each of those tags add a different meaning to the text encapsulated within them. In order to use markup languages the users can write the text and add all the tags manually, either by fully writing them or by using some kind of text helpers such as text editors with IntelliSense¹ which can help diminish the errors caused by manually writing the tags. But even with text helpers the resulting document can violate the restrictions of the respective markup language because the editors don't actually enforce the language rules.

The most common markup language is the HTML language, which is heavily used in Web applications. Other uses of the HTML language are writing emails, writing reports, etc. The function of HTML in Web applications is to define the user-interface, which is also known as the view of the website. To generate the end view the most common solution is the usage of *template engine* solutions, which can be considered the *controller* which is responsible for joining the domain data with the views. This approach separates the *view* from the *controller*,

¹[Intellisense Definition](#)

which allows both of the different layers of a project to be more independent. But, with the usage of *template engines*, there is another layers of complexity between the *view* and the *controller*, since both of these aspects of the project use different programming languages. In the following examples we will show the Java code necessary to generate a table, using the HTML language and the Mustache template engine.

In the following HTML example there is a violation of HTML rules, a `<html>` tag containing a `<div>` tag, which is not allowed (Listing 2.1).

```
1 <html>
2   <div>
3     <!-- (...) -->
4   </div>
5 </html>
```

Listing 2.1: Failed HTML rule validation

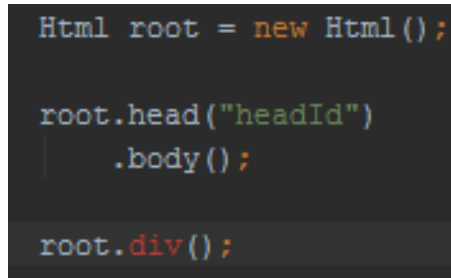
The solution to having documents that respect the markup language rules is changing the way the document writing works. As long as the writing process is fully controlled by the programmer errors will always happen because even though most text editors present the errors its corrections depend on the programmer correcting them. The suggestion presented is to move the writing control to an entity which can enforce the markup languages restrictions. This way it is guaranteed that the programmer won't be able to produce a document with errors.

Moving from the HTML representation to a Java representation of the issue there are multiple solutions to solve it. In this first code sample the programmer is allowed to add any child to the `Html` element (Listing 2.2), resulting in a violation of the language restrictions, which is not detected unless the programmer verifies the result manually.

```
1 Html root = new Html();
2 root.add(new Div());
```

Listing 2.2: Lack of rule validation

To solve this problem the entity, `Html` class, should restrict the children and attributes that accepts based on the existing restrictions on the HTML5 specification. As shown in the Figure 2.1 the generated API enforces the language restrictions in compile time, this way guaranteeing that any document generated will be compliant with the respective specification.



```
Html root = new Html();

root.head("headId")
    .body();

root.div();
```

Figure 2.1: Error validation in compile time

This solution even though apparently solving the main problem introduces a new one. The new problem resides on the fact that manually recreating all the rules of a given markup language is normally a very long process since most markup languages have a vast number of elements, attributes and restrictions.

The solution for this new problem is automation. An automated process that converts the markup language definition of elements, its attributes and restrictions to classes that represent those elements and methods which enforce the specification rules. With this automated process the application can generate a fluent API that allow the users to write their texts in a fluent way without errors and respecting the markup language specification. This is the main objective of this work, creating an infrastructure that reads a markup language definition, from a XSD file, and generates an API that allows the users to write well formed documents.

2.2 Use case

The use case that will be used to test and evaluate the solution will be the HTML5 XSD. In this case there are multiple elements that share behavior and/or attributes that can be generated automatically. The generated classes allow to create a tree of elements that represent a XML document. The resulting tree of elements can then be processed in different ways by using the Visitor pattern. This way each different Visitor implementation can use the generated tree of elements to write HTML documents to a stream, a socket or a file.

The generated HTML5 elements API will then be used in the HtmlFlow API, which is also a fluent Java API that is used to write well formed HTML files. With the Visitor pattern the HtmlFlow will only need to implement its own Visitor to achieve its goal. At the moment the HtmlFlow library only supports a set of the HTML elements which were created manually and the rest of the library interacts with those elements in order to write the HTML files. By using the

solution which will be developed in this work the HtmlFlow will support the whole HTML syntax.



State of Art

In this chapter we are going to introduce the XSD language in order to provide a better understanding of the next chapters and also introduce some tools that were discovered during the development of this dissertation.

3.1 XSD Language

The XSD language is a description of a type of XML document. Its purpose is to create a set of rules and constraints that a given type of XML document must follow in order to be considered valid. These rules are meant to create a contract on the type of information contained in the XML documents, apart from having well formed XML. To describe the rules and restrictions for a given XML document the XSD language relies on two types of data, elements and attributes. Elements are the most complex data type, they can contain other elements as children and can also have attributes. Attributes on the other hand are just pairs of information, defined by their name and their value. The value of a given attribute can be then restricted by multiple constraints existing on the language. There are multiple elements and attributes present in the XSD language, which are specified at [XSD Schema](#). In this dissertation we will use the set of rules and restrictions of the XSD files provided to build a fluent API that will enforce the rules and restrictions specified by the given file.

3.2 Template Engines

Templates engines are systems based on template files. A template file contains two types of information:

1. Static - This information doesn't change and should always be present.
2. Dynamic - This information depends on external input.

By using template files, the template engines are able to separate the presentation aspect of a given project from its business logic. This is usually important since by having these aspects separated posterior changes to the visual presentation will be implemented faster since there are very few dependencies between the project different layers. By using this approach template engines benefit on the following points:

1. Performance - By reusing the template files the engine avoids repetitive tasks.
2. Deployment Speed - It becomes easier to create new outputs, such as new web pages, fill a document, etc.

Template engines have been around for a long time, such as *JavaServer Pages* (JSP)¹ used with the Java language back in 1999, but gained a new significance with its application on the world of Web technologies. Multiple solutions have appeared that run the substitution process either on the client side of the web sites or all the way on the server side.

In the Sections 3.2.2 and 3.2.3 we introduce two solutions, J2html and Apache Velocity, which may be used in a similar way to `xml.et`.

3.2.1 Xmllet Similarities

Even though `xml.et` doesn't aim to be a template engine, both technologies share a few similarities. What similarities does `xml.et` share with most template engines?

¹[JavaServer Pages](#)

- Template definition - `xml.et` can define a template within the Java language, removing the need to use external files which usually implies using an extra "language" or syntax in order to define the template. This template definition has the disadvantage of being tied to a concrete language such as Java, which shouldn't be problematic if a project isn't supposed to change language.
- Template substitution - `xml.et` can also define a template to receive a certain type of data in order to fill the dynamic information of the template. This should be faster than the regular behaviour of template engines since `xml.et` doesn't need to read its template from a file on the file system.

3.2.2 J2html

J2html² is a Java library used to write HTML, a very similar solution to the used case presented in Section 2.2. The main difference between the two solutions are that the J2html does not verify the specification rules of the HTML language either at compile time or at runtime. This library also shows that the issue we are trying to solve with this dissertation is relevant since this library has quite a few forks and watchers on their github page³. In Chapter 5 we will present some performance tests to verify if our solution is more efficient at writing HTML.

3.2.3 Apache Velocity

Apache Velocity⁴ is a template engine that we discovered through J2html. Even though the `xml.et` solution doesn't define itself as a template engine it can also be used to such extent. Since the template engines are based on a template file, with some sort of code embedded in the language (HTML for example) the same result can be obtained by using the solution presented in this project. This solution improves the template engine solutions by allowing the users to define the exact same aspects defined in the template files directly into code, allowing the verification of the "template" at compile time by the language compiler. This reduces the overall complexity by removing possible errors in the template files and removing the necessity to separate template files and actual application code, while

²J2html

³J2html Github Page

⁴Apache Velocity

enforcing the language specification. In Chapter 5 we will also compare this solution with the J2html and the `xmlet` solution.

4

Solution

This chapter will present the `xmlet` solution, its different components and how they interact between them. Generating a Java API based on a XSD file includes two distinct tasks:

1. Parsing the information from the XSD file;
2. Generating the API based on the resulting information of the previous task.

Those tasks are encompassed by two different projects, `XsdParser` and `XsdAsm`. In this case the `XsdAsm` has a dependency to `XsdParser`.

4.1 XsdParser

`XsdParser` is a library that parses a XSD file into a list of Java objects. Each different XSD tag has a corresponding Java class and the attributes of a given XSD type are represented as fields in Java. All these classes derive from the same abstract class, `XsdAbstractElement`. All Java representations of the XSD elements follow the schema definition for XSD elements, referred in Section 3.1. For example, the `xsd:annotation` tag only allows `xsd:appinfo` and `xsd:documentation` as children nodes, and can also have an attribute named `id`, therefore `XsdParser` has the following class as shown in Listing 4.1.

```

1 public class XsdAnnotation extends XsdIdentifierElements {
2     //The id field is inherited from XsdIdentifierElements.
3     private List<XsdAppInfo> appInfoList = new ArrayList<>();
4     private List<XsdDocumentation> documentations = new ArrayList<>();
5
6     // (...)
7 }

```

Listing 4.1: XsdAnnotation class (Simplified)

4.1.1 Parsing Strategy

The first step of this library is handling the XSD file. The Java language has no built in library that parses XSD files, so we needed to look for other options. The main libraries found that address this problem were *Document Object Model* (DOM) and *Simple Application Programming Interface for eXtensive Markup Language* (SAX). After evaluating the pros and cons of those libraries the choice ended up being DOM, since a XSD file is a tree of XML elements. This choice was based mostly on the fact that SAX is an event driven parser and DOM is a tree based parser, which is more adequate for the present issue. DOM is a library that maps HTML, *eXtensive HyperText Markup Language* (XHTML) and XML files into a tree structure composed by multiple elements, also named nodes. This is exactly what XsdParser requires to obtain all the information from the XSD files, which is described in XML.

This means that XsdParser uses DOM to parse the XSD file and obtain its root element, a `xs:schema` node, performing a single read on the XSD file, avoiding multiple reads which is less efficient (Listing 4.2).

```

1 private Node getSchemaNode(String filePath)
2     throws IOException, SAXException, ParserConfigurationException {
3     DocumentBuilderFactory dbFactory =
4         DocumentBuilderFactory.newInstance();
5     DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
6     Document doc = dBuilder.parse(xsdFile);    //Parses the XSD file.
7
8     //Obtains the first node, which is the xs:schema node.
9     return doc.getFirstChild();
10 }

```

Listing 4.2: DOM Document Parsing

After obtaining the root node of the XSD file the XsdParser verifies if that node is a XsdSchema node as shown in Listing 4.3. If that is the case it proceeds by performing the parse function of the XsdSchema class.

```

1 Node schemaNode = getSchemaNode(filePath);
2
3 if (isXsdSchema(schemaNode)) {
4     XsdSchema.parse(this, schemaNode);
5 }

```

Listing 4.3: XsdParser Node Parsing Process

The XsdSchema element parse function converts the Node attributes into a Map object, which XsdSchema receives in the constructor. Each class extracts their field information from that Map object in their setFields method (Listing 4.4).

```

1 public class XsdSchema extends XsdAnnotatedElements {
2     private XsdSchema(XsdParser parser, Map<String, String> fieldsMap) {
3         super(parser, fieldsMap);
4     }
5     @Override
6     public void setFields(Map<String, String> fieldsMap) {
7         super.setFields(fieldsMap);
8
9         this.attributeFormDefault = fieldsMap.getOrDefault(
10             ATTRIBUTE_FORM_DEFAULT, "unqualified");
11         this.elementFormDefault = fieldsMap.getOrDefault(
12             ELEMENT_FORM_DEFAULT, "unqualified");
13         this.blockDefault = fieldsMap.getOrDefault(BLOCK_DEFAULT, "");
14         this.finalDefault = fieldsMap.getOrDefault(FINAL_DEFAULT, "");
15         this.targetNamespace = fieldsMap.getOrDefault(TARGET_NAMESPACE,
16             targetNamespace);
17         this.version = fieldsMap.getOrDefault(VERSION, version);
18         this.xmlns = fieldsMap.getOrDefault(XMLNS, xmlns);
19     }
20
21     public static ReferenceBase parse(XsdParser parser, Node node) {
22         NamedNodeMap nodeAttributes = node.getAttributes();
23         Map<String, String> attrMap = convertNodeMap(nodeAttributes);
24         XsdSchema schema = new XsdSchema(parser, attrMap);
25
26         return xsdParseSkeleton(node, schema);
27     }
28 }

```

Listing 4.4: XsdSchema Information Extraction (Simplified)

The parsing of the `XsdSchema` continues by parsing its children nodes. To parse children elements of any given `XsdAbstractElement` type we have the `xsdParseSkeleton` function present in the `XsdAbstractElement` class (Listing 4.5). This function will iterate in all the children of a given node, invoke the respective `parse` function of each children and then notify the parent element, using the Visitor pattern, so that the parent can perform the changes needed based on the type of element received. By using this strategy just by invoking the `parse` function of `XsdSchema`, which is the top element of all the XSD files, we effectively parse the whole document since each `parse` function of any `XsdAbstractElement` concrete type also invokes the `xsdParseSkeleton` function, which parses all the children.

```

1 ReferenceBase xsdParseSkeleton(Node node, XsdAbstractElement element){
2     XsdParser parser = element.getParser();
3     Node child = node.getFirstChild();
4
5     while (child != null) { //Iterates in all children from node.
6         //Only parses element nodes, ignoring comments and text nodes.
7         if (child.getNodeType() == Node.ELEMENT_NODE) {
8             String nodeName = child.getNodeName();
9
10            //Searches on a mapper for a parsing functions
11            //for the respective type.
12            BiFunction<XsdParser, Node, ReferenceBase> parserFunction =
13                XsdParser.getParseMappers().get(nodeName);
14
15            //Applies the parsing functions, if any, and notifies
16            //the parent objects Visitor to the newly created object.
17            if (parserFunction != null){
18                parserFunction.apply(parser, child).getElement()
19                    .accept(element.getVisitor());
20            }
21
22            child = child.getNextSibling(); //Moves on to the next sibling.
23        }
24
25        ReferenceBase wrappedElement= ReferenceBase.createFromXsd(element);
26        parser.addParsedElement(wrappedElement);
27        return wrappedElement;
28    }

```

Listing 4.5: XsdParseSkeleton Parsing Children From a Node

Based on the explanation provided above, we will give a more detailed description about the parsing process made by `XsdParser` using a small concrete example extracted from the HTML XSD file, present in Listing 4.6.

```

1 <xs:schema xmlns='http://schemas.microsoft.com/intellisense/html-5'
2     version="1.0"
3     targetNamespace='http://schemas.microsoft.com/intellisense/
      html-5'
4     xmlns:xs="http://www.w3.org/2001/XMLSchema">
5
6     <xs:element name="html">
7         <xs:complexType>
8             <!-- -->
9         </xs:complexType>
10    </xs:element>
11 </xs:schema>

```

Listing 4.6: Parsing Concrete Example

Step 1 - DOM parsing:

The parsing starts with the DOM library parsing the code (Listing 4.6), which returns the `xs:schema` node (i.e. `schemaNode` in Listing 4.3). `XsdParser` verifies if the node is in fact a `xs:schema` node and after verifying that in fact it is, it invokes the `XsdSchema parse` function (line 19 of Listing 4.4).

Step 2 - XsdSchema Attribute Parsing:

The `XsdSchema parse` function receives the `Node` object and converts it to a `Map` object (line 21 of Listing 4.4). The `map` object is then passed to the `XsdSchema` constructor which will result in the invocation of the `setFields` method (line 7 of Listing 4.4), which will extract the information from the `Map` object to the class fields.

Step 3 - XsdSchema Children:

To parse the `XsdSchema` children the `XsdAbstractElement xsdParseSkeleton` (Listing 4.5) function is called (line 24 of Listing 4.4) and starts to iterate the `xs:schema` node children, which, in this case, is a node list containing a single element, the `xs:element` node.

Step 4 - XsdElement Attribute Parsing:

The parsing of the `xs:element` node is similar to `xs:schema`, it extracts the attribute information from its respective node in its `setFields` function.

Step 5 - XsdSchema Visitor Notification:

After parsing the `xs:element` node the previously created `XsdSchema` object is notified. This notification informs the `XsdSchema` object that it contains the newly created `XsdElement` object using the Visitor pattern. The `XsdSchema` should then act accordingly based on the type of the object received as his children, since different types of objects should be treated differently.

This whole behaviour is shared by all the classes that represent a XSD element. The Visitor pattern is a very important tool in the parsing process since it allows each element to define a different behavior for each element received as children. This is also useful to implement the schema rules, since it can be used to define empty methods to reject any children that a given element type shouldn't contain as per definition on the schema specification. The same happens to the attributes present in the parsed DOM nodes, each `XsdAbstractElement` concrete type only extracts the attributes defined in the XSD schema specification, ignoring other attributes present.

4.1.2 Reference solving

After the parsing process described previously, there is still an issue to solve regarding the existing references in the XSD schema definition. In XSD files the usage of the `ref` attribute is frequent to avoid repetition of XML code. This generates two main problems when handling reference solving, the first one being existing elements with `ref` attributes referencing non existent elements and the other being the replacement of the reference object by the referenced object when present. In order to effectively help resolve the referencing problem some wrapper classes were added. These wrapper classes contain the wrapped element and serve as a classifier for the wrapped element. The existing wrapper classes are as follow:

- `UnsolvedElement` - Wrapper class to each element that has a `ref` attribute.
- `ConcreteElement` - Wrapper class to each element that is present in the file.
- `NamedConcreteElement` - Wrapper class to each element that is present in the file and has a `name` attribute present.
- `ReferenceBase` - A common interface between `UnsolvedReference` and `ConcreteElement`.

Having these wrappers on the elements allow for a detailed filtering, which is helpful in the reference solving process. That process starts by obtaining all the `NamedConcreteElement` objects since they may or may not be referenced by an existing `UnsolvedReference` object. The second step is to obtain all the `UnsolvedReference` objects and iterate them to perform a lookup search on the `NamedConcreteElement` objects obtained previously. This is achieved by comparing the value present in the `UnsolvedReference` `ref` attribute with the `NamedConcreteElement` `name` attribute. If a match is found then `XsdParser` performs a copy of the object wrapped by the `NamedConcreteElement` and replaces the element wrapped in the `UnsolvedReference` object that served as a placeholder. A concrete example of how this process works is in Listing 4.7.

```

1 <xsd:schema xmlns='http://schemas.microsoft.com/intellisense/html-5'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
2
3   <!-- NamedConcreteType wrapping a XsdGroup -->
4   <xsd:group id="replacement" name="flowContent">
5       <!-- (...) -->
6   </xsd:group>
7
8   <!-- ConcreteElement wrapping a XsdChoice -->
9   <xsd:choice>
10       <!-- UnsolvedReference wrapping a XsdGroup -->
11       <xsd:group id="toBeReplaced" ref="flowContent"/>
12   </xsd:choice>
13 </xsd:schema>

```

Listing 4.7: Reference Solving Example

In this short example we have a `XsdChoice` element that contains a `XsdGroup` element with a `ref` attribute. When replacing the `UnsolvedReference` objects the `XsdGroup` with the `ref` attribute is going to be replaced by a copy of the already parsed `XsdGroup` with the `name` attribute. This is achieved by accessing the parent of the element, in this case accessing the parent of the `XsdGroup` with the `ref` attribute, in order to remove the element identified by "toBeReplaced" and adding the element identified by "replacement".

Having created these classes it is expected that at the end of a successful file parsing only `ConcreteElement` and/or `NamedConcreteElement` objects remain. In case there are any remainder `UnsolvedReference` objects the programmer can query the parser, using the function `getUnsolvedReferences` of the `XsdParser` class, to discover which elements are missing and where were

they used. The programmer can then correct the missing elements by adding them to the XSD file and repeat the parsing process or just acknowledge that those elements are missing.

4.2 XsdAsm

XsdAsm is a library dedicated to generate a fluent Java API based on a XSD file. It uses the previously introduced XsdParser library to parse the XSD file contents into a list of Java elements that XsdAsm will use to obtain the information needed to generate the correspondent classes. To generate classes this library also uses the ASM¹ library, which is a library that provides a Java interface to bytecode manipulation, which provide method for creating classes, methods, etc. There were other alternatives to the ASM library but most of them are simply libraries that were built on top of ASM to simplify its usage. It supports the creation of Java classes up until Java 9 and is still maintained, the most recent version, 6.1, was release in 11 march of 2018. ASM also has some tools to help the new programmers understand how the library works. These tools help the programmers to learn faster how the code generation works and allow to increase the complexity of the generated code.

4.2.1 Supporting Infrastructure

To support the foundations of the XSD language an infrastructure is created in every API generated by this project. This infrastructure is composed by a common set of classes. This supporting infrastructure is divided into three different groups of classes:

Element classes:

- Element - An interface that serves as a base to every parsed XSD element.
- AbstractElement - An abstract class from where all the XSD element derive. This class implements most of the methods present on the Element interface.

Attribute classes:

¹[ASM Website](#)

- **Attribute** - An interface that serves as a base to every parsed XSD attribute.
- **BaseAttribute** - A class that implements the **Attribute** interface and adds restriction verification to all the deriving classes. All the attributes that have restrictions should derive from this class.

Visitor classes:

- **ElementVisitor** - An interface that defines methods for all the generated elements that can be visited with the Visitor pattern.
- **AbstractElementVisitor** - An abstract class that implements **ElementVisitor**. All the implemented methods point to a single method. This behaviour aims to reduce the amount of code needed to create concrete implementations of Visitors.

Taking in consideration those classes, a very simplistic API could be represented with the class diagram (Figure 4.1). In this example we have an element, **Html**, that extends **AbstractElement** and an attribute, **AttrManifestString**, that extends **BaseAttribute**.

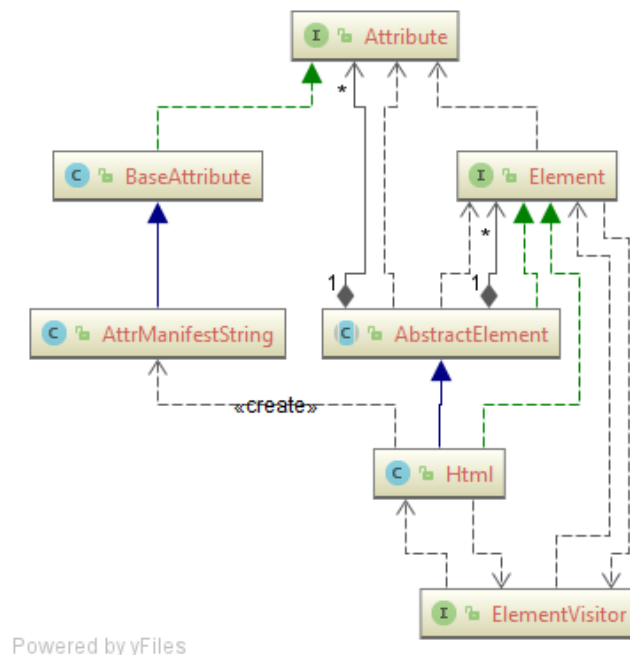


Figure 4.1: API - Supporting Infrastructure

4.2.2 Code Generation Strategy

To understand how most of this project works a XSD example (Listing 4.8) and a detailed explanation is provided. In this example there will be some simplifications making it easier to understand how the library works internally.

```
1 <xs:element name="html">
2
3   <xs:attributeGroup name="commonAttributeGroup">
4     <xs:attribute name="someAttribute" type="xs:string">
5   </xs:attributeGroup>
6
7   <xs:complexType>
8     <xs:choice>
9       <xs:element ref="body"/>
10      <xs:element ref="head"/>
11    </xs:choice>
12    <xs:attributeGroup ref="commonAttributeGroup" />
13    <xs:attribute name="manifest" type="xs:string" />
14  </xs:complexType>
15
16 </xs:element>
```

Listing 4.8: Code Generation XSD Example

With this example there is a multitude of classes that need to be created, apart from the always present supporting infrastructure presented in Section 4.2.1.

- **Html Element** - A class that represents the `Html` element (Listing 4.9), deriving from `AbstractElement`.
- **Body and Head Methods** - Both methods present in the `Html` class (Listing 4.9) that add `Body` (Listing 4.10) and `Head` (Listing 4.11) instances to `Html` children list.
- **Manifest Method** - A method present in `Html` class (Listing 4.9) that adds an instance of the `Manifest` attribute (Listing 4.12) to the `Html` attribute list.

```

1 public class Html extends AbstractElement implements
    CommonAttributeGroup {
2     public Html() { }
3
4     public Html attrManifest(String attrManifest) {
5         this.addAttr(new AttrManifest(attrManifest));
6     }
7
8     public Body body() { this.addChild(new Body()); }
9
10    public Head head() { this.addChild(new Head()); }
11 }

```

Listing 4.9: Html Element Class

- Body and Head classes - Classes for both Body (Listing 4.10) and Head (Listing 4.11) elements, similar to the generated Html class (Listing 4.9). The class contents will be dependent on the contents present in the concrete `xsd:element` nodes.

```

1 public class Body extends AbstractElement {
2     //Similar to Html, based on the contents of the respective
3     //xsd:element node.
4 }

```

Listing 4.10: Body Element Class

```

1 public class Head extends AbstractElement {
2     //Similar to Html, based on the contents of the respective
3     //xsd:element node.
4 }

```

Listing 4.11: Head Element Class

- Manifest Attribute - A class that represents the Manifest attribute (Listing 4.12), deriving from BaseAttribute.

```

1 public class AttrManifestString extends BaseAttribute<String> {
2     public AttrManifestString(String attrValue) {
3         super(attrValue);
4     }
5 }

```

Listing 4.12: Manifest Attribute Class

- **CommonAttributeGroup Interface** - An interface with default methods that add the group attributes to the concrete element (Listing 4.13).

```
1 public interface CommonAttributeGroup extends Element {  
2     default Html attrSomeAttribute(String attributeValue) {  
3         this.addAttr(new SomeAttribute(attributeValue));  
4         return this;  
5     }  
6 }
```

Listing 4.13: CommonAttributeGroup Interface

As we can see from the previous example this solution focus on how the code is organized instead of making complex code. All the methods present in the generated classes have very low complexity, mainly adding information to the element children and attribute list. To reduce repeated code many interfaces with default methods are created so different classes can implement them and reuse the code. The complexity of the generated code is mostly present in the `AbstractElement` class, which implements most of the `Element` interface methods. Another very important aspect of the generated classes is the extensive use of type arguments which allows the API to navigate in the element tree while maintaining type information which is essential to guarantee the specific language restrictions.

4.2.2.1 Type Arguments

As this solution was designed an objective became clear, the generated API should be easily navigable. This is crucial to provide a good user experience while creating templates through the `xmlet` APIs. So, there are two main aspects, the API should be easily navigable and always implement the concrete language restrictions. To tackle this issue we rely on type arguments. Through type arguments we can always keep track of the tree structure of the elements that are being created and keep adding elements or moving up in the tree structure. In Listing 4.14 we can observe how the type arguments work.

```

1 Html<Element> html = new Html<>();
2 Body<Html<Element>> body = html.body();
3 Div<Body<Html<Element>>> div = body.div();
4 Div<Body<Html<Element>>> divAfterAttribute =
5     div.attrClass("attrClassValue");
6 Body<Html<Element>> bodyAgain = divAfterAttribute.°();
7 Html<Element> htmlAgain = bodyAgain.°();

```

Listing 4.14: Explicit Example of Type Arguments

When we create the `Html` element we should indicate that he has a parent, for consistency. Then, as we add elements such as `Body` we automatically return the recently created `Body` element, but with parent information that indicates that this `Body` instance is descendant of an `Html` element. The same happens with the `Div` element. This behavior changes when adding attributes, in which case we return the object which had the attribute added to them, while maintaining all the parent information. The last two lines show how to move up in the element tree by using the method `°`. The method name is meant to be as short as possible and to avoid distractions while reading the template.

While in the example presented in Listing 4.14 the usage of the API might seem to have excessive verbose to define a simple HTML document that verbose isn't exactly needed. For specific purposes it might be needed to extract variables, e.g. to perform a loop and add some repeated table rows to a table, resulting in a code similar to the Listing 4.14 but the most common usage of the API should be more similar to Listing 4.15.

```

1 new Html<>()
2     .body()
3     .div() .attrClass("attrClassValue") .°()
4     .°();

```

Listing 4.15: Simpler Example of Type Arguments

To provide a better understanding on how this is possible we need to showcase three distinct classes. First we have the `AbstractElement` class, Listing 4.18, from which all concrete elements derive. This class receives two type arguments:

- **T** - Represents the type of the concrete element;
- **Z** - Represents the type of the parent of the concrete element.

In the `parent` method, which returns the parent of any concrete element, the type parameter is guaranteed by returning `Z`, which is the type of the parent of the current element, as shown in the last two lines of code in Listing 4.14.

```

1 class AbstractElement<T extends Element, Z extends Element>
2     protected Z parent;
3
4     public Z parent() {
5         return this.parent;
6     }
7
8     // (...)
9 }

```

Listing 4.16: Type Arguments - AbstractElement class

The second class is the `Html` class, which is representative of every concrete element of an `xmllet` API. It has a single type argument, `Z` which represents the type of its parent. It extends `AbstractElement` and therefore indicates that its type is `Html<Z>` and its parent type is `Z`. Any interface implemented by the concrete elements should receive the same type information as the `AbstractElement` class, as shown with `HtmlChoice0`. Regarding the `attrManifest` method, it indicates that returns the exact same instance as the one from which the method is called, keeping the type information by returning `Html<Z>`.

```

1 class Html<Z extends Element> extends AbstractElement<Html<Z>, Z>
2     implements HtmlChoice0<Html<Z>, Z>
3
4     // (...)
5
6     public Html<Z> attrManifest(String attrManifest) {
7         return this.addAttr(new AttrManifestString(attrManifest));
8     }
9 }

```

Listing 4.17: Type Arguments - AbstractElement class

As the third class we have the `HtmlChoice0` interface which is representative of most interfaces of an `xmllet` API. It should receive the same type arguments as `AbstractElement`:

- **T** - Represents the type of the concrete element;
- **Z** - Represents the type of the parent of the concrete element.

Since the `Html` is the only class implementing this interface its `T` type parameter is always `Html<Z>` which results in a return type of `Body<Html<Z>>` when the method `body` is called.

```

1 interface HtmlChoice0<T extends Element<T, Z>, Z extends Element>
                                extends Element<T, Z> {
2     default Body<T> body() {
3         return (Body)this.addChild(new Body(this.self()));
4     }
5 }

```

Listing 4.18: Type Arguments - AbstractElement class

4.2.3 Restriction Validation

In the description of any given XSD file there are many restrictions in the way the elements are contained in each other and which attributes are allowed. To reflect those restrictions to Java language there are two alternatives, validation in runtime or in compile time. This library tries to validate most of the restrictions in compile time, as shown above by the way classes are created. But some restrictions can't be validated in compile time, an example of this is the following restriction (Listing 4.19):

```

1 <xs:schema>
2     <xs:element name="testElement">
3         <xs:complexType>
4             <xs:attribute name="intList" type="valuelist"/>
5         </xs:complexType>
6     </xs:element>
7
8     <xs:simpleType name="valuelist">
9         <xs:restriction>
10             <xs:maxLength value="5"/>
11             <xs:minLength value="1"/>
12         </xs:restriction>
13         <xs:list itemType="xsd:int"/>
14     </xs:simpleType>
15 </xs:schema>

```

Listing 4.19: Restrictions Example

In this example (Listing 4.19) we have an element (i.e. `testElement`) that has an attribute called `intList`. This attribute has some restrictions, it is represented

by a `xs:list`, the list elements have the `xsd:int` type and its element count should be between 1 and 5. Transporting this example to the Java language will result in the following class (Listing 4.20):

```
1 public class AttrIntList extends BaseAttribute<List> {  
2     public AttrIntList(List<Integer> list) {  
3         super(list);  
4     }  
5 }
```

Listing 4.20: Attribute Class Example

But with this solution the `xs:maxLength` and `xs:minLength` values are ignored. To solve this problem the existing restrictions in any given attribute are hardcoded in the class static constructor, which stores the restrictions in a static Map object as showed in Listing 4.21.

```
1 static {  
2     restrictions = new ArrayList<Map<String, Object>>();  
3     HashMap<String, Object> restriction = new HashMap<>();  
4     restriction.put("MaxLength", Integer.valueOf(5));  
5     restriction.put("MinLength", Integer.valueOf(1));  
6     restrictions.add(restriction);  
7 }
```

Listing 4.21: Attribute Static Constructor Restrictions

By using this strategy the restrictions can be validated whenever an instance of a concrete attribute is created. To enforce the restrictions present in the Map object the `BaseAttribute` constructor (Listing 4.22) will pass those values to a class, `RestrictionValidator`, which validates all the different types of restrictions, in this case `xs:maxlength` and `xs:minlength`. Each different restriction has its validation method (i.e. `validateMaxLength` and `validateMinLength` at Listing 4.23 lines 6 and 11, respectively) which will throw an exception if the value (i.e. `val`) to validate does not match the restriction. By using this strategy the API ensures that any successful usage follows the rules previously defined by the schema.

```
1 public class BaseAttribute<T> implements Attribute<T> {
2     static List<Map<String, Object>> restrictions = new ArrayList();
3
4     public BaseAttribute(T var1, String var2) {
5         // ...
6         restrictions.forEach(this::validateRestrictions);
7     }
8
9     private void validateRestrictions(Map<String, Object> restriction){
10         Object value = this.getValue();
11
12         if (value instanceof String) {
13             RestrictionValidator.validate(restriction, (String)value);
14         }
15
16         if (value instanceof Integer || value instanceof Short ||
17             value instanceof Float || value instanceof Double) {
18             RestrictionValidator.validate(restriction, (Double)value);
19         }
20
21         if (value instanceof List) {
22             RestrictionValidator.validate(restriction, (List)value);
23         }
24     }
25 }
```

Listing 4.22: BaseAttribute Rule Validation Restrictions

```

1 public class RestrictionValidator {
2     static void validate(Map<String, Object> restMap, List val) {
3         validateMinLength(restMap.getDefault("MinLength", -1), val);
4         validateMaxLength(restMap.getDefault("MaxLength", -1), val);
5     }
6     private static void validateMaxLength(int maxLength, List list) {
7         if (maxLength != -1 && list.size() > maxLength) {
8             throw new RestrictionViolationException("Violation of
9                 maxLength restriction.");
10        }
11    }
12    private static void validateMinLength(int minLength, List list) {
13        if (minLength != -1 && list.size() < minLength) {
14            throw new RestrictionViolationException("Violation of
15                minLength restriction.");
16        }
17    }
18 }

```

Listing 4.23: Restriction Validator Class (Simplified)

4.2.3.1 Enumerations

Regarding restrictions there is one that can be enforced at compile time, the `xs:enumeration`. To obtain that validation at compile time the XsdAsm library generates Enum classes that contain all the values indicated in the `xs:enumeration` tags. In the following example (Listing 4.24) we have an attribute with three possible values, command, checkbox and radio.

```

1 <xs:attribute name="type">
2     <xs:simpleType>
3         <xs:restriction base="xsd:string">
4             <xs:enumeration value="command" />
5             <xs:enumeration value="checkbox" />
6             <xs:enumeration value="radio" />
7         </xs:restriction>
8     </xs:simpleType>
9 </xs:attribute>

```

Listing 4.24: Enumeration XSD Definition

This results in the creation of an Enum, EnumTypeCommand (Listing 4.25). The attribute class will then receive an instance of EnumTypeCommand, ensuring that only allowed values are used (Listing 4.26).

```

1 public enum EnumTypeCommand {
2     COMMAND (String.valueOf("command")),
3     CHECKBOX (String.valueOf("checkbox")),
4     RADIO (String.valueOf("radio"))
5 }

```

Listing 4.25: Enumeration Class

```

1 public class AttrTypeEnumTypeCommand extends BaseAttribute<String> {
2     public AttrTypeEnumTypeCommand(EnumTypeCommand attrValue) {
3         super(attrValue.getValue());
4     }
5 }

```

Listing 4.26: Attribute Receiving An Enumeration

4.2.4 Element Binding

To support repetitive tasks over an element the Element and AbstractElement classes were modified to support binders. This allows programmers to define, for example, templates for a given element. An example is presented in Listing 4.27 using the HTML5 API.

```

1 public class BinderExample{
2     public void bindExample(){
3         Html<Html> root = new Html<>();
4         Body<Html<Html>> body = root.body();
5
6         Table<Body<Html<Html>>> table = body.table();
7         table.tr().th().text("Title");
8         table.<List<String>>binder((elem, list) ->
9             list.forEach(tdValue ->
10                 elem.tr().td().text(tdValue)
11             )
12         );
13         //Keep adding elements to the body of the document.
14     }
15 }

```

Listing 4.27: Binder Usage Example

In this example we create a table and add a title in the first row as a title header (i.e. `th()`). In regard to the values present in the table instead of having them inserted right away it is possible delay that insertion by indicating what will the element do when the information is received. This is achieved by implementing a Visitor that supports binding.

In Listing 4.28 we can observe how the Visitor would work. It maintains the default behaviour on the elements that aren't bound (i.e. `else` clause). In the case that the element is bound to a function this implementation will clone the element and apply a model (i.e. a `List<String>` object following the example of Listing 4.27) to the clone, effectively executing the function supplied in the previously called `binder` method (i.e. Listing 4.27 line 8). This function call will generate new children on the cloned table element which will be iterated as if they belonged to the original element tree. This behaviour ensures that the original element tree isn't affected since all these changes are performed in a clone of the bound element, meaning that the template can be reused.

```
1 public <T extends Element> void sharedVisit(Element<T, ?> element) {
2     // ...
3
4     if(element.isBound()) {
5         List<Element> children = element.cloneElem()
6                                     .bindTo(model)
7                                     .getChildren();
8         children.forEach( child -> child.accept(this));
9     } else {
10        element.getChildren().forEach(item -> item.accept(this));
11    }
12
13    // ...
14 }
```

Listing 4.28: Visitor with binding support

4.3 Client

To use and test both XsdAsm and XsdParser we need to implement a client for XsdAsm. Two different clients were implemented, one using the HTML5 specification and another using the specification for Android visual layouts. In this section we are going to explore how the HTML5 API is generated using the XsdAsm library and how to use the resulting API.

4.3.1 HtmlApi

To generate the HTML5 API we need to obtain its XSD file. After that there are two options, the first one is to create a Java project that invokes the XsdAsm main method directly by passing the path of the specification file and the desired API name (Listing 4.29).

```
1 void generateApi(String filePath, String apiName){  
2     XsdAsmMain.main(new String[] {filePath, apiName} );  
3 }
```

Listing 4.29: API creation

The second option is using the Maven² build lifecycle³ to make that same invocation by adding an extra execution to the *Project Object Model* (POM) file (Listing 4.30) to execute a batch file that invokes the XsdAsm main method (Listing 4.31).

²Maven

³Maven Build Lifecycle

```

1 <plugin>
2   <artifactId>exec-maven-plugin</artifactId>
3   <groupId>org.codehaus.mojo</groupId>
4   <version>1.6.0</version>
5   <executions>
6     <execution>
7       <id>create_classes1</id>
8       <phase>validate</phase>
9       <goals>
10        <goal>exec</goal>
11      </goals>
12      <configuration>
13        <executable>
14          ${basedir}/create_class_binaries.bat
15        </executable>
16      </configuration>
17    </execution>
18  </executions>
19 </plugin>

```

Listing 4.30: Maven API compile classes plugin

```

1 if exist ".\src/main/java" rmdir ".\src/main/java" /s /q
2
3 if not exist ".\target/classes/org/xmllet/htmlapi"
4   mkdir ".\target/classes/org/xmllet/htmlapi"
5
6 call
7   mvn exec:java -D"exec.mainClass"="org.xmllet.xsdasm.main.XsdAsmMain"
8   -D"exec.args"=". \src/main/resources/html_5.xsd htmlapi"

```

Listing 4.31: Maven API creation batch file (create_class_binaries.bat)

This client uses the Maven lifecycle option by adding an execution at the validate phase (Listing 4.30, line 8) which invokes XsdAsm main method to create the API. This invocation of XsdAsm creates all the classes in the target folder of the HtmlApi project. Following these steps would be enough to allow any other Maven project to add a dependency to the HtmlApi project and use its generated classes as if they were manually created. But this way the source files and Java documentation files are not created since XsdAsm only generates the class binaries. To tackle this issue we added another execution to the POM. This execution uses the Fernflower⁴ decompiler, the Java decompiler used by IntelliJ⁵ *Integrated*

⁴Fernflower Decompiler

⁵IntelliJ IDE

Development Environment (IDE), to decompile the classes that were automatically generated (Listing 4.32, 4.33).

```

1 <!-- Plugin Information -->
2 <execution>
3   <id>decompile_classes</id>
4   <phase>validate</phase>
5   <goals>
6     <goal>
7       exec
8     </goal>
9   </goals>
10  <configuration>
11    <executable>
12      ${basedir}/decompile_class_binaries.bat
13    </executable>
14  </configuration>
15 </execution>

```

Listing 4.32: Maven API decompile classes plugin

```

1 if not exist "./src/main/java/org/xmllet/htmlapi" mkdir "./src/main/java
  /org/xmllet/htmlapi"
2
3 call
4   mvn exec:java
5   -D"exec.mainClass"="org.jetbrains.java.decompiler.main.decompiler.
    ConsoleDecompiler"
6   -D"exec.args"="-dgs=true ./target/classes/org/xmllet/htmlapi ./src/
    main/java/org/xmllet/htmlapi"
7
8 if exist "./target/classes/org" rmdir "./target/classes/org" /s /q

```

Listing 4.33: Maven API decompile batch file (decompile_class_binaries.bat)

By decompiling those classes we obtain the source code which allows us to delete the automatic generated classes and allow the Maven build process to perform the normal compiling process, which generates the Java documentation files and the class binaries, along with the source files obtained from the decompilation process. This process, apart from generating more information to the programmer that will use the API in the future, also allows to find any problem with the generated code since it forces the compilation of all the classes previously generated.

4.3.1.1 Using the HtmlApi

After the previously described compilation process of the HtmlApi project we are ready to use the generated API. To start using it the first step is to implement a custom Visitor class, which defines what to do when the created element tree is visited. A very simple example is presented in Listing 4.34 which writes the HTML tags based on the name of the element visited and navigates in the element tree by accessing the children of the current element.

```

1 public class CustomVisitor<R> implements ElementVisitor<R> {
2
3     private PrintStream printStream = System.out;
4
5     public CustomVisitor(){ }
6
7     public <T extends Element> void sharedVisit(Element<T, ?> element)
8     {
9         printStream.printf("<%s", element.getName());
10
11         element.getAttributes()
12             .forEach(attribute ->
13                 printStream.printf(" %s=\"%s\"",
14                     attribute.getName(), attribute.getValue()));
15
16         printStream.print(">\n");
17
18         element.getChildren().forEach(item -> item.accept(this));
19
20         printStream.printf("</%s>\n", element.getName());
21     }
22 }
```

Listing 4.34: Custom Visitor

After creating the Visitor presented in Listing 4.34 we can start to create the element tree that we want convert to text using the CustomVisitor. To start we should create a Html object, since all the HTML documents have it as a base element. Upon creating that root element we can start to add other elements or attributes that will appear as options based on the specification rules. To help with the navigation on the element tree a method was created to allow the navigation to the parent of any given element. This method is named `parent`, a short method name to keep the code as clean as possible. In Listing 4.35 we can see a code example that uses a good amount of the API features, including element creation and how

they are added to the element tree, how to add attributes, attributes that receive enumerations as parameters and how to navigate in the element tree using the method `°`.

```

1 Html<Html> root = new Html<>();
2
3 root.head()
4     .meta() .attrCharset ("UTF-8") .°()
5     .title()
6         .text ("Title") .°()
7     .link() .attrType (EnumTypeContentType.TEXT_CSS)
8         .attrHref ("/assets/images/favicon.png") .°()
9     .link() .attrType (EnumTypeContentType.TEXT_CSS)
10        .attrHref ("/assets/styles/main.css") .°() .°()
11 .body() .attrClass ("clear")
12     .div()
13         .header()
14             .section()
15                 .div()
16                     .img() .attrId ("brand")
17                         .attrSrc ("./assets/images/logo.png") .°
18                             ()
19                     .aside()
20                         .em()
21                             .text ("Advertisement")
22                             .span()
23                                 .text ("HtmlApi is great!");
24
25 CustomVisitor customVisitor = new CustomVisitor();
26 customVisitor.visit(root);

```

Listing 4.35: HtmlApi Element Tree

With this element tree presented (Listing 4.35) and the previously presented `CustomVisitor` (Listing 4.34) we obtain the following result (Listing 4.36). The indentation was added for readability purposes, since the `CustomVisitor` implementation in Listing 4.34 does not indent the resulting HTML.

```
1 <html>
2   <head>
3     <meta charset="UTF-8">
4   </meta>
5   <title>
6     Title
7   </title>
8   <link type="text/css" href="/assets/images/favicon.png">
9   </link>
10  <link type="text/css" href="/assets/styles/main.css">
11  </link>
12 </head>
13 <body class="clear">
14   <div>
15     <header>
16       <section>
17         <div>
18           
19         </img>
20         <aside>
21           <em>
22             Advertisement
23           <span>
24             HtmlApi is great!
25           </span>
26         </em>
27       </aside>
28     </div>
29   </section>
30 </header>
31 </div>
32 </body>
33 </html>
```

Listing 4.36: HtmlApi Visitor Result

The `CustomVisitor` of Listing 4.36 is a very minimalist implementation since it does not indent the resulting HTML, does not simplify elements with no children (i.e. the `link`/`img` elements) and other aspects that are particular to HTML syntax. That is where the `HtmlFlow` library come in, it implements the particular aspects of the HTML syntax in its `Visitor` implementation which deals with how and where the output is written.



Deployment

5.1 Github Organization

This project and all its components belong to a github organization called `xmlet`¹. The aim of that organization is to contain all the related projects to this dissertation. All the generated APIs are also created as if they belong to this organization.

5.2 Maven

In order to manage the developed projects a tool for project organization and deployment was used, named Maven. Maven has the goal of organizing a project in many different ways, such as creating a standard of project building and managing project dependencies. Maven was also used to generate documentation and deploying the projects to a central code repository, Maven Central Repository². All the releases of projects belonging to the `xmlet` Github organization can be found under the groupId, [com.github.xmlet](https://search.maven.org/artifact/com.github.xmlet).

¹[xmlet Github](#)

²[Maven Central Repository](#)

5.3 Sonarcloud

Code quality and its various implications such as security, low performance and bugs should always be an important issue to a programmer. With that in mind all the projects contained in the `xml.et` solution were evaluated in various metrics and the results made public for consultation. This way, either future users of those projects or developers trying to improve the projects can check the metrics as another way of validating the quality of the produced code. The tool to perform this evaluation was Sonarcloud³, which provides free of charge evaluations and stores the results which are available for everyone. Sonarcloud also provides an API to show badges that allow to inform users of different metrics regarding a project. Those badges are presented in the `xml.et` modules Github pages, as shown in Figure 5.1.



Figure 5.1: XsdParser Github badges

5.4 Testing metrics

To assert the performance of the `xml.et` solution we used the HTML5 use case to compare it against the J2html (Section 3.2.2) and Apache Velocity (Section 3.2.3), presented earlier. This two libraries have a difference that is crucial when dealing with performance, J2html doesn't indent the generated HTML while Apache Velocity does it. In order to perform a fair comparison between solutions two Visitors were used with the HTML5 solution, one that indents the HTML to compare it with Velocity and other that doesn't indent it to compare it to J2html. The

³[Sonarcloud xml.et page](#)

computer used to perform all the tests present in this section has the following specs:

Processor: Intel Core i3-3217U 1.80GHz

RAM: 4GB

The tests that are presented below consist in the generation of a simple HTML document, with a table and a variable amount of table entries as shown in Listing 5.1.

```
1 <html>
2   <body>
3     <table>
4       <tr>
5         <th>
6           Title
7         </th>
8       </tr>
9
10      <!-- Repeated based on the number of elements -->
11      <tr>
12        <td>
13          ElemX
14        </td>
15      </tr>
16
17    </table>
18  </body>
19 </html>
```

Listing 5.1: Test HTML

The textual elements to feature in the table data rows are stored in the same data structure and each solution should iterate that structure when generating the expected HTML. To assert which solution is the fastest each one of them will generate the same HTML document and we will verify the number of iterations that are possible to perform in a certain unit of time. To achieve this objective we will use *Java Microbenchmark Harness (JMH)*⁴, which is a tool used for benchmarking. The values gathered are result of the mean values of 8 testing iterations after 12 iterations of warm up.

⁴[JMH Website](#)

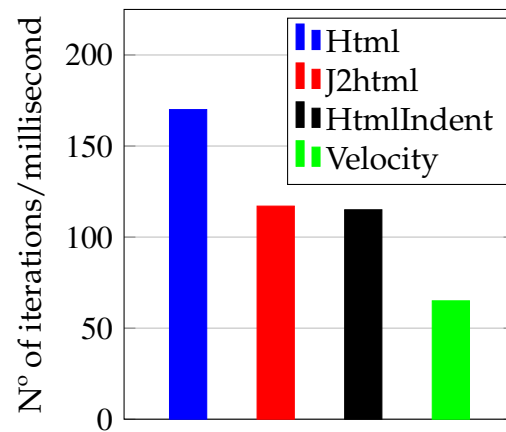


Figure 5.2: Benchmark: Generating an Html Table with 10 Elements

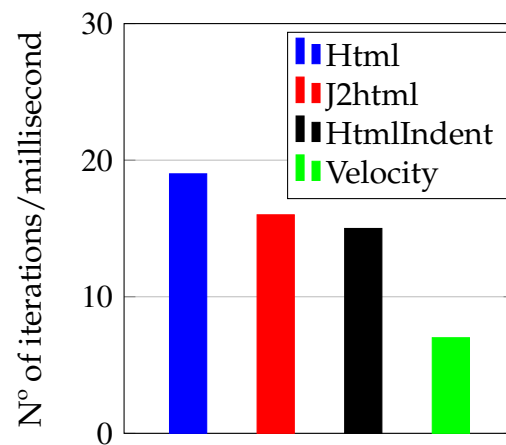


Figure 5.3: Benchmark: Generating an Html Table with 100 Elements

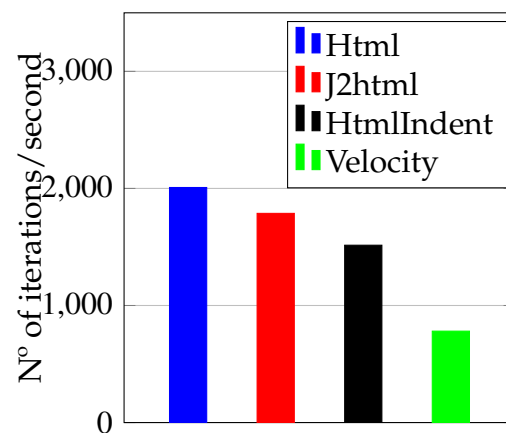


Figure 5.4: Benchmark: Generating an Html Table with 1.000 Elements

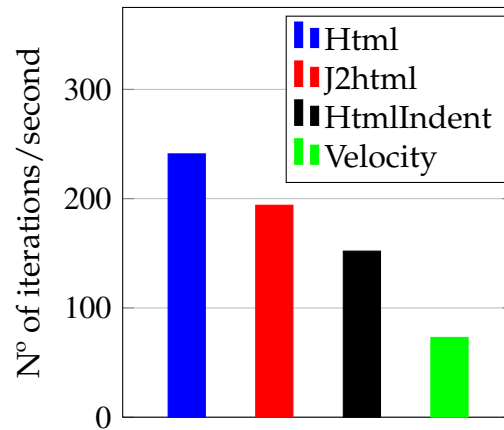


Figure 5.5: Benchmark: Generating an Html Table with 10.000 Elements

As we can see by the presented results the `HtmlApi` outperforms the other solutions, either with few elements or with a large number of elements.

5.4.1 Spring benchmark

To assert that the results obtained through the previous benchmark weren't in any way biased in favor of the `HtmlApi` we performed another benchmark. This second benchmark was performed by using a preexisting project that compares ten different template engines such as Jade⁵, Mustache⁶, Pebble⁷, Velocity⁸, Chunk⁹, JSP¹⁰, Jtwig¹¹, FreeMarker¹², Thymeleaf¹³, Handlebars¹⁴. This benchmark, named `spring-comparing-template-engines`¹⁵, uses a tomcat server¹⁶ to launch a WebApi based on the Java Spring¹⁷ framework. This WebApi provides routes for each solution to test, e.g. `http://localhost:8080/mustache`, which returns a standard HTML page based in two distinct aspects.

⁵Jade Template Engine

⁶Mustache Template Engine

⁷Pebble Template Engine

⁸Velocity Template Engine

⁹Chunk Template Engine

¹⁰JSP Template Engine

¹¹Jtwig Template Engine

¹²FreeMarker Template Engine

¹³Thymeleaf Template Engine

¹⁴Handlebars Template Engine

¹⁵Template Engines Benchmark

¹⁶Apache Tomcat

¹⁷Java Spring

- Template - Each solution under benchmark defines the same template, in their own syntax. An example is the Mustache template in Listing 5.2.
- Data - A List of Presentation objects (Listing 5.3), which all the solutions receive to complete their templates.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8"/>
5     <meta name="viewport" content="width=device-width, initial-
      scale=1.0"/>
6     <meta http-equiv="X-UA-Compatible" content="IE=Edge"/>
7     <title>{{#i18n}}example.title{/i18n}} - Mustache</title>
8     <link rel="stylesheet" href="{{rc.contextPath}}/webjars/
      bootstrap/3.3.7-1/css/bootstrap.min.css"/>
9   </head>
10  <body>
11    <div class="container">
12      <div class="page-header">
13        <h1>{{#i18n}}example.title{/i18n}} - Mustache</h1>
14      </div>
15      {{#presentations}}
16        <div class="panel panel-default">
17          <div class="panel-heading">
18            <h3 class="panel-title">{{title}} - {{
              speakerName}}</h3>
19          </div>
20          <div class="panel-body">
21            {{{summary}}}
22          </div>
23        </div>
24      {{/presentations}}
25    </div>
26    <script src="{{rc.contextPath}}/webjars/jquery/3.1.1/jquery.min
      .js"></script>
27    <script src="{{rc.contextPath}}/webjars/bootstrap/3.3.7-1/js/
      bootstrap.min.js"></script>
28  </body>
29 </html>

```

Listing 5.2: Spring Benchmark Mustache Template

```
1 public class Presentation {  
2     private Long id;  
3     private String title;  
4     private String speakerName;  
5     private String summary;  
6     private String room;  
7     private Date startTime;  
8     private Date endTime;  
9 }
```

Listing 5.3: Spring Benchmark Presentation Object

Since all the different solutions used in this API generate the same HTML document using the same information we can evaluate which solution is faster. To incorporate the `HtmlApi` solution in this `WebApi` we registered another entry in Spring settings and defined the same template as the other solutions.

After setting the Spring project with our solution we need to introduce other tool, the `ApacheBench`¹⁸. This tool is used to flood a given *Uniform Resource Locator* (URL) with requests and measures how much time it takes for the server to respond to those requests. The goal is to flood every route with a set number of requests and evaluate the time that each solution needs to respond to the same number of requests, effectively measuring each solution performances. As a last step we also evaluated the overhead that the Spring framework introduces in this process by creating a route that returns an empty answer. By obtaining this overhead value we can subtract it to the results obtained in each benchmark route in order to exclusively compare the time that the respective solution needs to generate their response.

The benchmark values presented below are the result of a single iteration after running two warm up iterations. Multiple variants were used to test the template engines and the `HtmlApi` solution. The presented results are obtained after running the specified benchmark and *removing* the Spring overhead.

¹⁸[ApacheBench](#)

10.000 requests using 10 threads with a Spring overhead of 1.031 seconds:

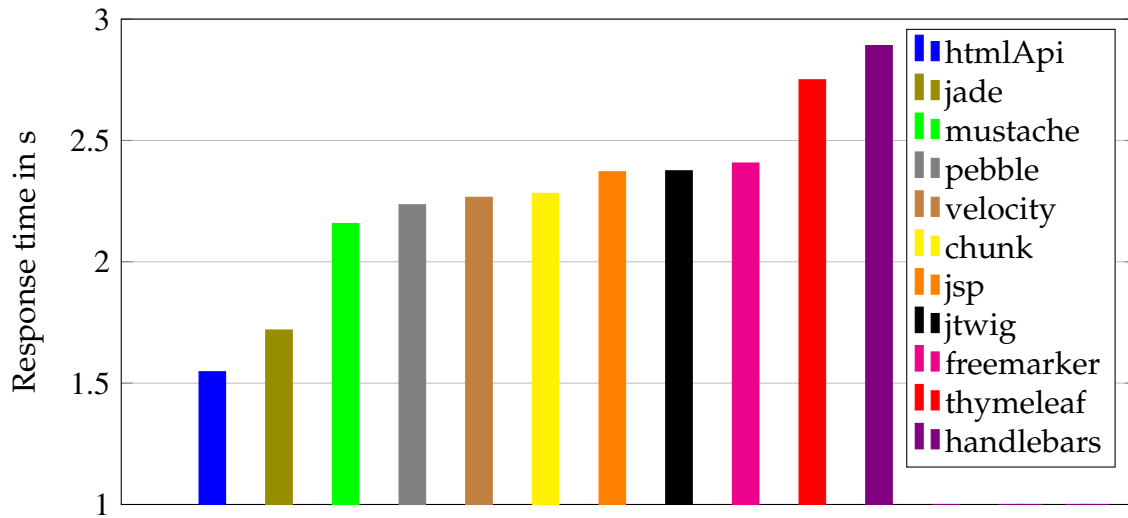


Figure 5.6: Benchmark: Spring Benchmark with 10.000 requests

30.000 requests using 10 threads with a Spring overhead of 3.285 seconds:

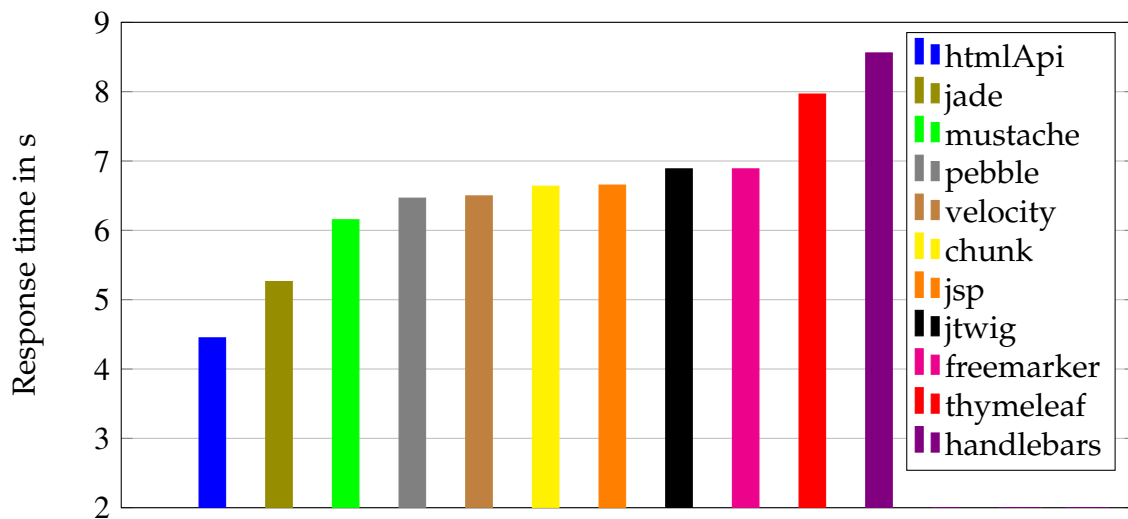


Figure 5.7: Benchmark: Spring Benchmark with 30.000 requests issued with 10 threads

30.000 requests using 25 threads with a Spring overhead of 3.787 seconds:

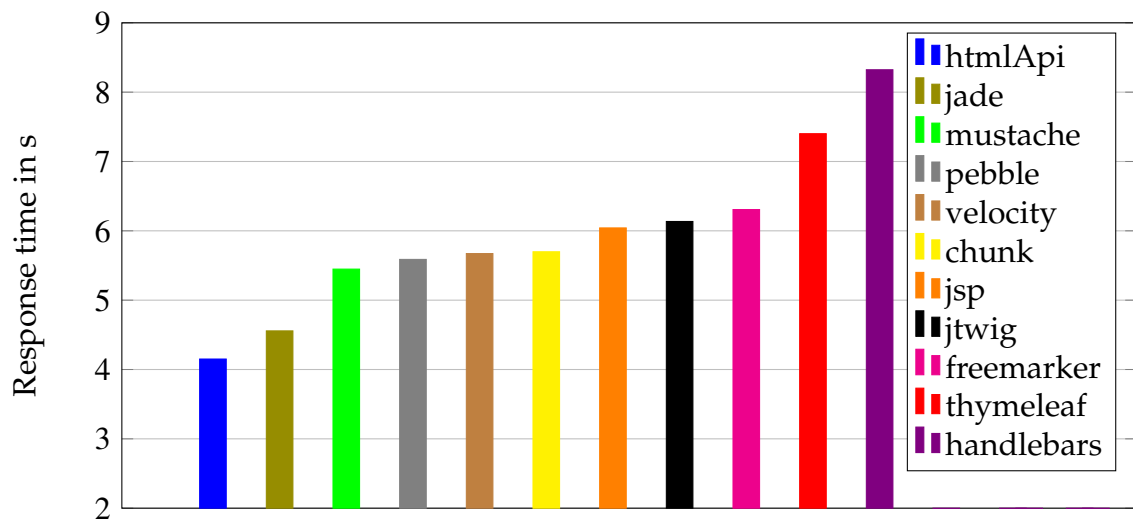


Figure 5.8: Benchmark: Spring Benchmark with 30.000 requests issued with 25 threads

100.000 requests using 10 threads with a Spring overhead of 9.890 seconds:

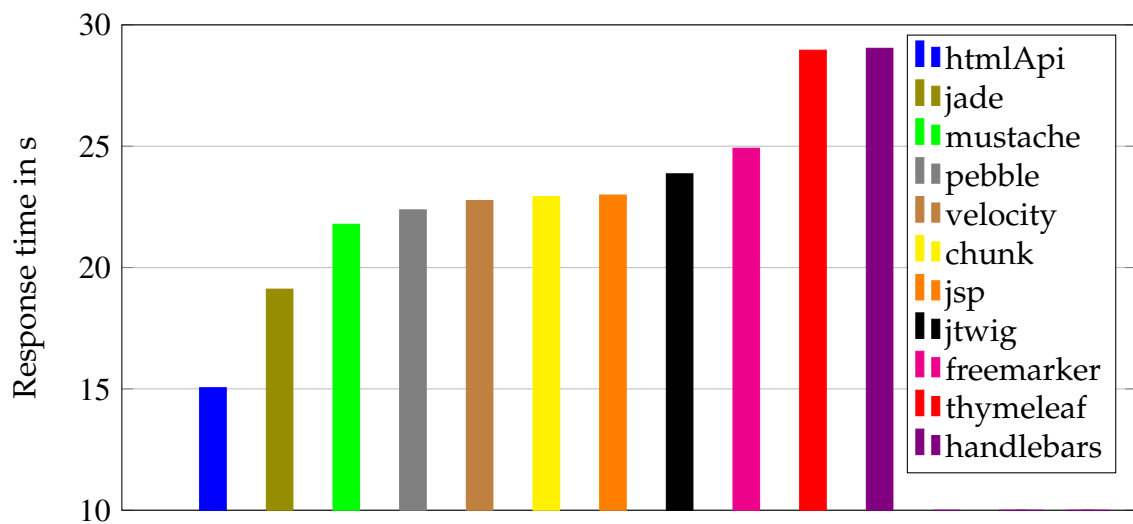


Figure 5.9: Benchmark: Spring Benchmark with 100.000 requests

As the results show, HtmlApi outperforms all the template engines in the examples presented with the gap in performance increasing as the number of elements grows. With the two examples presented with 30.000 requests we can observe

that the results are approximately the same, even though the concurrent request count increased from 10 to 25, with some template engines performing a bit better and other a bit worse, which may be due to invariable execution variations.



Conclusion

In this dissertation we developed a structure of projects that can interpret a XSD document and use its contents to generate a fluent Java API that allows to perform actions over the language defined in the XSD document while enforcing most of the rules that exist in the XSD syntax. The generated API only reflects the structure described in the XSD document, providing tools that allow any future usage to be defined according to the needs of the user. Upon testing the resulting solution we obtained better results than similar solutions while proving a solution with a fluent language, which should be intuitive even for people that never programmed in Java before, since it ends up being very similar to writing XML.

The main language definition used in order to test and develop this solution was the HTML5 syntax, which generated the HtmlApi project, containing a set of classes reflecting all the elements and attributes present in the HTML language. This HtmlApi project was then used by the HtmlFlow library in order to provide an API that writes well formed HTML documents. Other XSD files were used to test the solution, such as the Android layouts definition file, which defines the existing XML elements used to create visual layouts for the Android operating system and the attributes that each element contains.

6.1 Future work

In the future there are already changes planed to the way that the APIs are generated. The current APIs generated by the present solution are built on a two step basis, first the user has to create the element tree and after finishing its creation the user can then visit the whole tree. Even though this solution provides security to the user and already provides better results than similar solutions there is an improved solution that should increase its performance in a significant way. The improved solution removes the two steps nature of the solution, avoiding storing elements in a data structure and then having to iterate that structure. An example of this new solution usage is shown in Listing 6.1.

```

1 Visitor visitor = new Visitor();
2 Html<Html> documentRoot = new Html<>(visitor);
3 documentRoot.body();

```

Listing 6.1: Future Solution Usage 1

With the root receiving the Visitor instance its possible to start calling the respective visit methods right away, as shown in class Html in Listing 6.2 and Body in Listing 6.3.

```

1 public class Html{
2     private Visitor visitor;
3
4     public Html(Visitor visitor){
5         this.visitor = visitor;
6         visitor.visit(this);
7     }
8
9     public Body body(){
10         return new Body(this.self() );
11     }
12
13     public Html someAttribute(String value){
14         visitor.visit(new SomeAttribute(value));
15         return this;
16     }
17 }

```

Listing 6.2: Future Solution Html Class (Simplified)

```

1 public class Body{
2     private Visitor visitor;
3
4     public Body(Element parent){
5         this.visitor = parent.getVisitor();
6         this.visitor.visit(this);
7     }
8 }

```

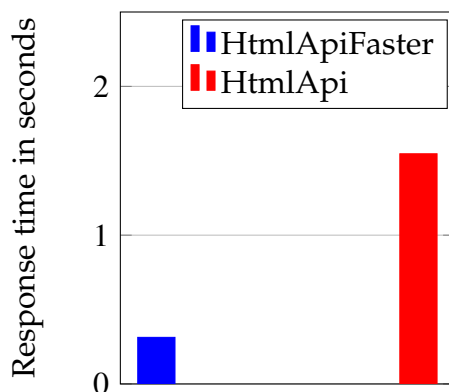
Listing 6.3: Future Solution Body Class (Simplified)

With this solution the overhead of using this solution is minimized, because as we can see the overhead of the code structure almost resumes itself to the overhead of instantiating classes. All the rules of the language are still enforced due to the way that the classes are organized and the usage keeps its fluent nature since the existent methods and its return type are unchanged. This also removes all the overhead of storing and iterating data from a data structure. Using this solution in the web world should improve the response times of servers in a very significant way since the solution can write to the response stream as soon as it executes the method code for adding an element, opposed to having to wait that the whole tree is built and only then iterated.

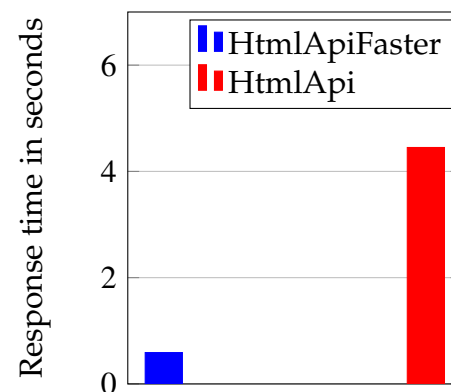
6.1.1 Early Results

After implementing a first version of this improved solution, called *HtmlApiFaster*, we ran a few tests in order to evaluate its performance. The benchmark used was the same presented in Section 5.4 with the *spring-comparing-template-engines* project. All the results presented below are presented without the respective Spring overhead, which was previously explained in Section 5.4.

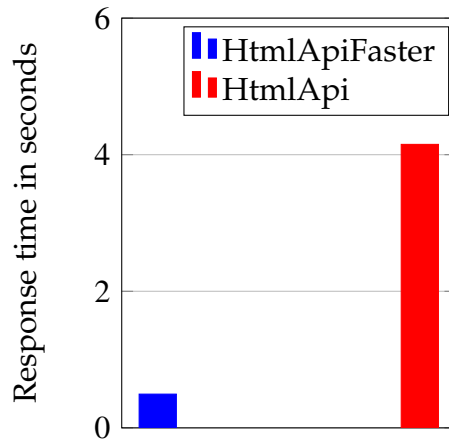
10.000 requests, 10 threads:



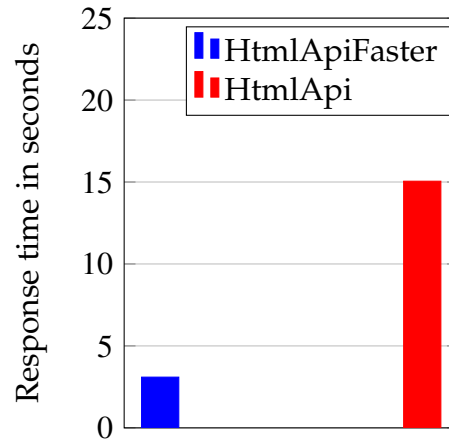
30.000 requests, 10 threads:



30.000 requests, 25 threads:



100.000 requests, 10 threads:



As we can see the performance gains are very good, with the lowest gap being with the example using 100.000 requests where HtmlApiFaster is 4 times faster than HtmlApi. On the other hand the biggest difference is present in the example using 30.000 requests and 25 concurrent threads, where HtmlApiFaster is 8 times faster than HtmlApi which was already the better option when comparing to the template engines. These results are promising and the HtmlApiFaster will be further developed in order to deploy it as an alternative to HtmlApi.