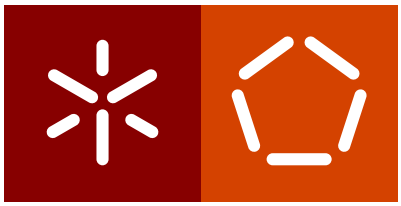
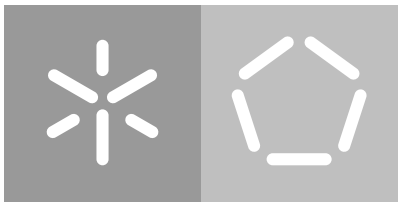


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hugo André Coelho Cardoso

Relatório de Pré-Dissertação

Fevereiro 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hugo André Coelho Cardoso

Relatório de Pré-Dissertação

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação realizada sob a orientação de
José Carlos Leite Ramalho

Fevereiro 2022

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração. Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

_____

RESUMO

O objetivo desta dissertação é o desenvolvimento de uma aplicação que permita gerar automaticamente *datasets* sintéticos representativos e, possivelmente, bastante extensos, a partir de *schemas* de *JSON* e *XML*, de forma a facilitar a testagem de aplicações de *software* e empreendimentos científicos em áreas como a *data science*.

Para esta finalidade, pretende-se desenvolver uma plataforma assente sobre o *DataGen*, uma aplicação que permite a prototipagem rápida de *datasets* através da sua própria *Domain Specific Language* (*DSL*) de especificação de modelos de dados. O *DataGen* é capaz de processar estes modelos e gerar posteriormente *datasets* sintéticos que obedeçam às restrições estruturais e semânticas estabelecidas, automatizando todo o processo de geração de dados com valores espontâneos gerados em tempo de execução e/ou provenientes de bancos de dados de suporte.

O produto a desenvolver chamar-se-á *DataGen From Schemas* e constituirá um complemento ao *DataGen*, operando conjuntamente com o mesmo e partilhando as mesmas camadas de lógica de negócio e de dados, de forma a assegurar a compatibilidade das plataformas e a portabilidade dos modelos criados entre ambas. O objetivo deste produto é expandir os casos de aplicação do *DataGen*, dando a opção de gerar dados sintéticos diretamente a partir de *schemas*. Visa atingir isto através da conversão de *schemas* para modelos da linguagem de domínio do *DataGen*, incorporando então a aplicação-base como uma espécie de *middleware* para gerar os *datasets*.

O relatório presente de pré-dissertação pretende detalhar a fase inicial de estudo do tema, tecnologias pertinentes e trabalho relacionado existente, bem como a fase de ideação da aplicação final, projetando uma arquitetura adequada e levantando requisitos para o *DataGen From Schemas*, tendo em conta as conclusões alcançadas através da pesquisa realizada.

Palavras-Chave: Schemas, JSON, XML, Geração de Dados, Dados Sintéticos, DataGen, DSL, Dataset, Gramática, Aleatoriedade, Open Source, Data Science, REST API, PEG.js

CONTEÚDO

ACRÓNIMOS

A

API Application Programming Interface.

C

CPU Central Process Unit.

D

DOM Document Object Model.

DSL Domain Specific Language.

H

HTTP Hypertext Transfer Protocol.

I

IOT Internet of Things.

J

JSON JavaScript Object Notation.

JWT JSON Web Token.

M

MVVM Model-View-ViewModel.

N

NOSQL Not Only SQL.

R

REST Representational State Transfer.

RNID Regulamento Nacional de Interoperabilidade Digital.

W

W3C World Wide Web Consortium.

X

XML Extensible Markup Language.

XSD XML Schema Definition.

1

INTRODUÇÃO

Este capítulo inicial serve para estabelecer o tema da presente dissertação, contextualizando-o na perspetiva atual de desenvolvimento de *software* e explicando a sua necessidade e possível aplicação num vasto leque de áreas científicas. De seguida, é explicada a motivação para o desenvolvimento deste projeto, os objetivos do mesmo e a metodologia sob a qual se desenrolará, acabando com um breve resumo da estrutura deste relatório e do que é tratado em cada capítulo.

1.1 CONTEXTO

No panomara atual do mercado tecnológico e de desenvolvimento de *software*, são cada vez mais exploradas áreas que operam com quantidades volumosas de dados, com o objetivo de extrair conhecimento e conclusões a partir da informação disponível. É o caso da *data science*, que visa aplicar métodos científicos de análise e algoritmia a bancos de dados para adquirir conhecimento novo, que pode posteriormente ser aplicado em várias áreas e para diversos fins. Outro exemplo é o *machine learning*, que procura usar este método para munir sistemas informáticos com a capacidade de aprendizagem automática, acedendo a dados e aprendendo através da sua análise, a fim de se tornarem mais eficientes na sua função.

Contudo, num mundo onde a necessidade de bancos de dados representativos e avultados aumenta a cada dia, políticas de proteção de dados e preocupações relacionadas com a privacidade e segurança dos utilizadores (Mostert et al., 2016) surgem como entraves ao desenvolvimento destas áreas e à progressão de muitos projetos para a fase de desenvolvimento (GAO, 2020) ou testagem. Além disso, a incapacidade de partilhar dados devido a estas preocupações vem também impedir empresas de procurar ajuda e/ou colaboração externa, em muitos casos (Patki et al., 2016).

Neste contexto, a geração de dados sintéticos surge como uma possível solução para estes problemas, sob a premissa de criar conjuntos de dados realistas e representativos artificialmente, através das especificações estruturais fornecidas. Esta abordagem foi originalmente proposta por Rubin em 1993 (Rubin, 1993), procurando explorar uma nova via que permitisse usar e partilhar dados sem desrespeitar as regulações rigorosas de tratamento

de dados sensíveis (como é o exemplo da *GDPR* da União Europeia (GDP, 2018)), uma vez que os dados, embora semelhantes aos que podem ser encontrados em bases de dados empresariais, não seriam obtidos por medição direta. Desde então, este método tem vindo a ser cada vez mais empreendido e refinado, aplicado nas mais diversas áreas como *Smart Homes* (Dahmen and Cook, 2019), modelos de microsimulação espacial (Smith et al., 2009), *Internet of Things* (*IoT*) (Anderson et al., 2014), *Deep Learning* (Ekbatani et al., 2017) e na área automotiva (Tsirikoglou et al., 2017).

Esta dissertação procura expor e documentar a ideação e desenvolvimento da aplicação *DataGen From Schemas*, cujo objetivo é a geração de dados sintéticos a partir de *schemas* de *XML* e *JSON*. Esta aplicação deve cumprir dois requisitos: por um lado, deve ser capaz de gerar *datasets* de tamanho significativo e com informação realista. Por outro lado, deve também conseguir processar os modelos introduzidos pelo utilizador e obedecer às suas especificações, de forma a que os dados gerados sejam formalmente e estruturalmente válidos face aos ficheiros em questão. De forma a cumprir estas condições, a plataforma será construída sobre outra aplicação já existente, *DataGen*, que é abordada em mais detalhe no capítulo 2.

1.2 Motivação

O *DataGen* é uma ferramenta versátil que permite a prototipagem rápida de *datasets* e a testagem de aplicações de software. Atualmente, esta solução é das poucas que oferece tanto a complexidade como a escalabilidade necessárias para gerar *datasets* adequados à avaliação do desempenho de uma *API* de dados ou de uma aplicação complexa, permitindo testar as mesmas com um volume apropriado de dados heterogéneos.

O cerne do *DataGen* é a sua *Domain Specific Language* (*DSL*) que foi criada para especificar *datasets*, tanto a nível estrutural como de conteúdo, e que é dotada de um vasto leque de mecanismos que permitem a geração de *datasets* bastante variados, lidando com requirimentos muito exigentes e intrincados.

O *DataGen From Schemas* surge como um complemento e uma extensão desta aplicação, procurando gerar *datasets* diretamente a partir de ficheiros de especificação *XML Schema* e *JSON Schema*, através das rotas aplicacionais *REST* expostas pelo *DataGen*. Desta forma, a criação de um modelo de especificação de *XML* ou *JSON*, que são vastamente populares e utilizados no mercado, surge como alternativa à escrita das regras operacionais na *DSL* do *DataGen* para definir o *dataset*, passando isto a ser um passo intermédio realizado em segundo plano.

Assim, a presente dissertação visa contribuir para o aumento significativo do número de casos de aplicação deste processo de geração, tornando o processo menos complexo e mais acessível para qualquer utilizador. Este novo componente age como uma camada de

abstração sobre a aplicação já existente, ignorando a necessidade de aprender a utilizar a *DSL* a partir da documentação e agilizando imenso o processo de especificação estrutural.

1.3 OBJETIVOS

O objetivo principal desta dissertação é a geração de datasets a partir de ficheiros de especificação (*Schemas*) dos formatos de troca de dados *XML* e *JSON*. Mais especificamente, pretende-se:

- Desenvolvimento de um compilador para filtrar e processar *XML Schemas* e outro para *JSON Schemas*;
- Criação de programas conversores capazes de gerar modelos válidos na *DSL* do *DataGen* a partir dos dados filtrados;
- Desenvolvimento de uma aplicação web, com características *user-friendly* e um fluxo de trabalho intuitivo;
- Criação de rotas *REST* para possibilitar a utilização da aplicação sem recurso à interface e eventual integração em terceiros;
- Estabelecimento de comunicação entre a nova aplicação e o *DataGen*;
- Testagem da implementação realizada;
- *Deployment* do *DataGen From Schemas*.

1.4 METODOLOGIA

A metodologia de trabalho para esta dissertação segue os seguintes passos:

- Estudo e pesquisa bibliográfica sobre geração de dados sintéticos a partir de modelos de especificação;
- Levantamento e análise de requisitos para a aplicação;
- Estudo das tecnologias mais adequadas para a implementação;
- Desenvolvimento da solução;
- Testagem e análise de desempenho da implementação;
- Reuniões semanais com o orientador.

1.5 Estrutura do documento

O presente capítulo visa apresentar o tema abordado por esta dissertação, procurando contextualizar a sua relevância e necessidade no mundo moderno, explicar a motivação inerente à proposta de exploração deste tópico, estabelecer os objetivos e resultados esperados do projeto e expor a metodologia para o seu desenvolvimento.

O capítulo 2 serve para expor o estado da arte, ou seja, procura apresentar todo o conhecimento obtido até à data sobre o tópico desta dissertação, no que toca a tecnologias pertinentes e ao trabalho relacionado existente. Começa pelo estudo dos formatos *JSON* e *XML*, de maneira a ganhar um entendimento claro do que se pretende gerar e de que dificuldades poderão surgir derivadas das suas diferenças. De seguida, realiza-se uma análise profunda à aplicação que servirá de base a esta, o *DataGen*, mostrando em que capacidades é adequada para esse papel e quais as suas lacunas que será necessário resolver. Por fim, efetua-se a investigação de alguns casos de estudo de aplicações semelhantes já existentes, procurando identificar as suas vantagens e pontos fracos, de forma a realizar um levantamento de requisitos minucioso para o *DataGen From Schemas*.

De seguida, no capítulo 3 é proposta uma abordagem detalhada e bem fundamentada para o desenvolvimento do produto final, postulando uma arquitetura compartimentalizada que partilha componentes com o *DataGen*, justificando a escolha das tecnologias indicadas e delineando claramente as funcionalidades que se pretende implementar em cada parte da aplicação.

Por fim, no capítulo 4 são apresentadas algumas conclusões sobre o trabalho realizado até este ponto e reportado neste relatório, refletindo sobre esta fase de pesquisa e ideação e o respetivo cumprimento de todos os requisitos objetivos antes de avançar para a fase de desenvolvimento.

2

ESTADO DA ARTE

Este capítulo pretende expor o conhecimento obtido no seguimento da investigação realizada até à data sobre o tema da dissertação e o trabalho relacionado existente. Primeiro, efetuar-se-á um estudo dos formatos *XML* e *JSON*, a fim de apurar o tipo de informação que conseguem reprensentar e que circunstâncias são mais adequadas para utilizar um ou outro, de maneira a ter um bom entendimento dos dados que se pretende fabricar. De seguida, proceder-se-á a uma análise minuciosa do *DataGen*, a aplicação-base sobre a qual vai assentar o *DataGen From Schemas*, de forma a analisar a sua compatibilidade com *schemas* e a definir as ferramentas das quais vai ser possível tirar proveito para o processo de tradução dos ficheiros. Por fim, realizar-se-á um estudo sobre o trabalho relacionado - neste caso, geradores de *XML* e *JSON* a partir dos respetivos ficheiros de especificação. O objetivo é realizar um levantamento de requisitos, procurando identificar as vantagens e pontos fracos dos produtos existentes, a fim de apurar uma lista de características desejáveis e a evitar para o *DataGen From Schemas*.

2.1 FORMATOS A ADAPTAR

O *Regulamento Nacional de Interoperabilidade Digital* (*RNID*), publicado em 2018, é uma norma que busca uniformizar a disponibilização de informação na *web* efetuada por sistemas informáticos em Portugal, estabelecendo que toda a informação existente em circulação deve ser proporcionada em formatos abertos (não proprietários), de forma a assegurar a interoperabilidade técnica e semântica, em termos globais, dentro da Administração Pública. Este regulamento, que reflete uma abordagem cada vez mais universal também no panorama da informática internacional (nomeadamente as diretrizes europeias em termos de interoperabilidade), levou a que a maioria dos *sites* tendesse para dois formatos em particular, tradicionalmente usados para a comunicação de informação entre serviços *web* - *JSON* e *XML*.

Em termos históricos, o *XML* começou por ser o formato mais popular para a troca de informação entre aplicações, no período inicial do surgimento e estabelecimento dos serviços *web*. Com o tempo, deduziu-se que era um formato muito pesado para esse

propósito, sendo que os ficheiros de informação produzidos se revelavam demasiado grandes e retardavam a comunicação entre aplicações e a própria prestação dos serviços, pelo que começaram a ser exploradas outras alternativas. Entre estas, o *JSON* relevou-se uma das mais interessantes e práticas, graças à sua natureza leve e compacta, passando a ser incrementalmente implementado nesta área, ao longo dos seguintes anos. Eventualmente, com o surgimento das arquiteturas *REST*, o *JSON* tornou-se no formato de eleição para transportar informação entre serviços, usurpando o lugar do *XML* e estabelecendo-se como o formato mais popular, embora o *XML* continue ainda hoje a ser bastante utilizado neste contexto.

Como tal, os formatos de dados que se pretende implementar no *DataGen From Schemas* são precisamente o *JSON* e o *XML* que, embora sejam considerados concorrentes na área de troca de informação, podem representar tipos de informação diferentes e ter aplicações distintas. Neste subcapítulo, conduzir-se-á uma análise a estes formatos, procurando entender as suas estruturas e o resultado que se pretende obter a partir das respetivos *schemas*. Pretende-se com isto identificar as principais diferenças entre as duas linguagens e prever os obstáculos que podem surgir à sua geração com recurso ao *DataGen*, como será discutido em mais detalhe na secção 2.2.

2.1.1 *JSON*

JavaScript Object Notation (*JSON*) é um formato leve de troca de dados derivado da linguagem de programação *JavaScript*, cuja simplicidade e legibilidade humana contribuíram para a sua vasta popularidade atual. Possui um amplo e crescente suporte em inúmeras linguagens de programação (*C++, Java, JavaScript, Python*, etc), apresentado-se como um forte candidato para a troca rápida e compacta de informação entre aplicações.

Este formato é usado para representar **informação estruturada**, isto é, dados com estruturas rígidas e bem-definidas, não permitindo quaisquer divergências da estrutura estabelecida na *schema*, nomeadamente tipos de dados diferentes. Isto reflete-se na sintaxe *JSON*, que é construída unicamente sobre duas estruturas de dados diferentes:

- **objetos** - coleções não ordenadas de pares chave-valor:

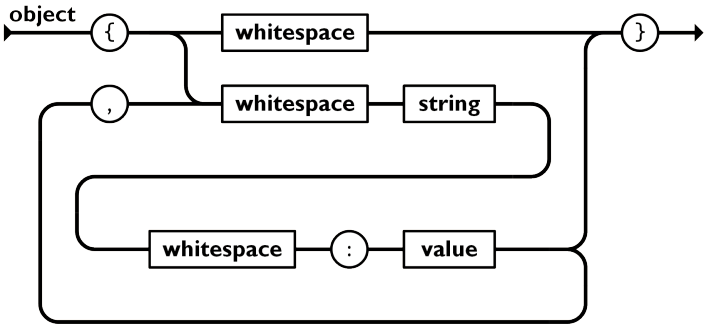


Figura 1: Estrutura de um objeto *JSON*

- **arrays** - listas ordenadas de valores:

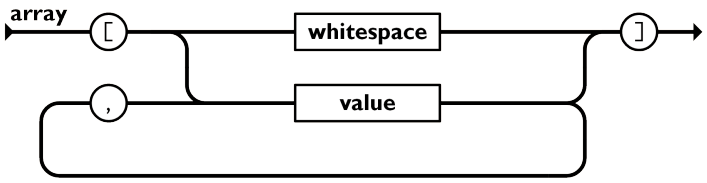


Figura 2: Estrutura de um *array JSON*

2.1.2 *XML*

Extensible Markup Language (*XML*) é um formato textual flexível que constitui uma **meta-linguagem** e pode servir de base a outras, servindo o propósito de publicação e troca de informação.

Este formato apresenta suporte tanto para **informação estruturada** como **semi-estruturada** - dados cujo modelo possui uma estrutura formal flexível, podendo apresentar diversos graus de liberdade, mas que contêm marcadores (neste caso, *tags*) para separar elementos semânticos e distinguir campos dentro dos dados. Segue-se um exemplo de dados semi-estruturados em *XML*, com a respetiva especificação *XSD*:

```
% Definição em XML Schema
<xs:element name="paragrafo">
    <xs:complexType mixed="true">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="pessoa" type="xs:string"/>
            <xs:element name="pais" type="xs:string"/>
            <xs:element name="cidade" type="xs:string"/>
            <xs:element name="ano" type="xs:string"/>
            <xs:element name="profissao" type="xs:string"/>
```

```
        </xs:choice>
    </xs:complexType>
</xs:element>

% Exemplo de elemento XML produzido
<paragraph>
  <pessoa>Albert Einstein</pessoa> (<ano>1879</ano>-<ano>1955</ano>) foi um
  <profissao>físico</profissao> alemão, nascido na cidade de
  <cidade>Ulm</cidade> em <ano>1879</ano>. Quando jovem, mudou-se para a
  <pais>Suíça</pais>, onde se formou, tornando-se <profissao>professor
  </profissao> da Escola Politécnica de <cidade>Zurique>/cidade>.
</paragraph>
```

Passando então ao estudo da sintaxe e à estrutura dos registos criados em *XML*, observa-se as seguintes características:

- faz uso de *tags* para separar o conteúdo da formatação, delimitando claramente onde começa e acaba o conteúdo de cada elemento;
- estrutura hierárquica de elementos - para cada documento, há um elemento-raiz e todos os outros encontram-se aninhados neste;
- cada elemento possui:
  - *tags* de abertura e de fecho, ou apenas uma, opcionalmente, se o conteúdo for vazio;
  - atributos: representam "características" do elemento, informação complementar do conteúdo;
  - conteúdo: pode ser simples (texto) ou complexo (aninhamento de outros elementos).

2.2 DATAGEN

O *DataGen* é uma aplicação *web full-stack*, grátis e *open-source*, disponível em `https://datagen.di.uminho.pt/`. Foi desenvolvido pelo autor desta dissertação, Hugo Cardoso, em colaboração com outros colegas e sob a mentoria do orientador José Carlos Ramalho, que posteriormente propôs expandir as funcionalidades da aplicação, a fim de ser capaz de gerar *datasets* diretamente a partir de *schemas*, ideia que viria a dar aso a esta dissertação e ao *DataGen From Schemas*.

O *DataGen* constitui uma ferramenta poderosa e versátil que permite gerar *datasets* volumosos e de conteúdo variado ao gosto do utilizador, a partir de especificações na sua própria *DSL*. Para nele basear o *DataGen From Schemas*, é preciso garantir primeiro que esta aplicação possui os meios necessários para gerar dados a partir de ficheiros de especificação. Por outras palavras, é necessário verificar que existe compatibilidade entre a informação representada nos *schemas* e as funcionalidades/método de geração do programa. O objetivo é usar o *DataGen* como um *middleware*: uma vez que esta aplicação já é capaz de gerar *datasets* a partir de modelos na sua própria *DSL*, pretende-se criar um programa que processe os *schemas* e os traduza para modelos desses, encarregando então o *DataGen* de produzir o resultado final.

Como tal, o *DataGen* deve cumprir vários requisitos:

1. Capacidade de gerar dados tanto em *JSON* como em *XML*;
2. Capacidade de gerar informação estruturada (*JSON*) e semi-estruturada (*XML*);
3. Capacidade de programar todos os tipos de dados primitivos das *schemas*;
4. Capacidade de gerar objetos recursivos;
5. Capacidade de geração difusa.

2.2.1 *Geração em JSON e XML*

O *DataGen* fornece a opção de gerar *datasets* tanto em *JSON* como em *XML*. Durante o processamento do modelo da *DSL*, o seu compilador gera uma estrutura de dados intermédia com o *dataset* final, constituída por um objeto *JSON*, que pode então ser traduzida para *XML* (Santos et al., 2021). Este processo de tradução cria uma *string* com os dados em *XML*, à medida que vai percorrendo a estrutura recursivamente. Isto significa que a tradução de *JSON schema* para a *DSL* e da *DSL* para *JSON* é direta, não havendo quaisquer complicações, uma vez que a estrutura intermédia gerada é também ela um objeto *JSON*. Logo, pode-se concluir que o *DataGen* é ideal para criar *datasets* de informação estruturada neste formato.

Contudo, a geração de dados em *XML* da aplicação é rudimentar: apenas produz elementos, possivelmente aninhados, com uma *tag* de abertura, o respetivo valor e uma *tag* de fecho. Isto deve-se ao facto de a aplicação ter sido inicialmente projetada para produzir dados em *JSON*, daí a sintaxe da linguagem específica de domínio ser inspirada nesse formato, tendo sido adicionada apenas posteriormente a tradução para *XML*. Este pormenor acarreta complicações para a tradução de *XML schema* para a *DSL*, devido a um conjunto fulcral de diferenças estruturais entre os dois formatos, que se traduz nas seguintes propriedades de *JSON*:

- não tem noção de atributos;
- define apenas conteúdo estruturado, não é capaz de representar conteúdo semi-estruturado como *XML*. Isto é, não consegue envolver uma propriedade em texto, também esse texto envolvente teria de ter uma chave para pertencer à estrutura *JSON*;
- não suporta várias propriedades com chave igual ao mesmo nível de profundidade, sobrescrevendo-as e vingando o último valor declarado, enquanto que isso é possível em *XML*;

De forma a contornar isto, será necessário expandir as funcionalidades do tradutor para *XML* do *DataGen*. No modelo produzido pelo *DataGen From Schemas*, nas circunstâncias supramencionadas e eventualmente noutras, enviar-se-ão sinalizadores especiais nas chaves das propriedades, que serão posteriormente processados de acordo com o seu significado no programa-tradutor, a fim de colmatar as lacunas do formato *JSON* para representação de informação em *XML*. Na tradução, o programa tratará de remover os sinalizadores e preservar apenas as chaves originais, realizando a formatação adequada a cada caso. Seguir-se-á uma convenção pouco legível e relativamente rebuscada, de forma a garantir que utilizadores da *frontend* do *DataGen* não usam estes sinalizadores por coincidência e obtêm resultados diferentes do pretendido:

- atributo: `DFS_ATTR__{nome_atributo}`;
- elemento semi-estruturado: `DFS_MIXED__{chave}`;
- as chaves dos elementos serão numeradas, mantendo um contador para cada chave diferente, de forma a não se sobrescreverem na estrutura intermédia em *JSON*: `DFS_1__{chave}`, `DFS_2__{chave}`, ...

Através deste método, tornar-se-á possível a representação de conteúdo semi-estruturado nos dados gerados, sendo que o programa-tradutor se encarregará de identificar os elementos com conteúdo *mixed* através dos respetivos sinalizadores e encapsulá-los com *lorem ipsum*. Logo, pode-se concluir que se cumpre o requisito 2.

Desta forma, o programa-tradutor ficará muito mais rico em termos de funcionalidades e será possível gerar *datasets* em *XML* bastante complexos e interessantes, verificando-se também o requisito 1 na sua totalidade.

2.2.2 *Geração dos tipos de dados primitivos*

Como foi clarificado no subcapítulo 2.1, os dois tipos de *schemas* considerados possuem diferentes tipos de dados primitivos. Para garantir que o *DataGen* é uma aplicação adequada

para servir de base a este projeto, é necessário garantir que é possível gerar cada um desses tipos, através das ferramentas da sua linguagem de domínio.

Além disso, é também pertinente e bastante relevante a opção de conseguir introduzir aleatoriedade controlada na geração dos valores, de forma a que o resultado não seja determinista. Ao usar o *DataGen* como *middleware*, pretende-se também fornecer o modelo intermédio da *DSL* ao utilizador, dando-lhe a opção de utilizar o modelo diretamente na *frontend* do *DataGen*, se assim pretender, e poder editá-lo ao seu gosto. Como tal, convém introduzir aleatoriedade controlada neste modelo, de forma a não produzir o mesmo resultado sempre que for processado, permitindo assim gerar *datasets* heterogéneos a partir do mesmo estado inicial.

JSON Schemas

O principal componente do *DataGen* é a sua *DSL*, que permite especificar o *dataset* pretendido, tanto a nível estrutural (aninhamento de campos, estruturas de dados) como a nível de conteúdo (tipos de dados, relações entre campos). A sintaxe desta linguagem assenta no formato *JSON*, tendo-lhe sido acrescentadas outras ferramentas incrementalmente, nomeadamente mecanismos de repetição, geração difusa, entre outros. Como tal, existe uma tradução direta do formato *JSON* para a *DSL*, pelo que incorpora todos os tipos de dados primitivos de *JSON*:

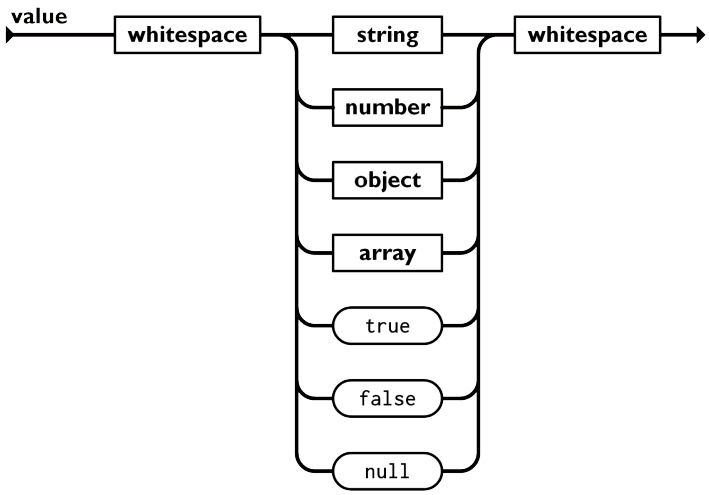


Figura 3: Tipos de dados primitivos do formato *JSON*

De seguida, é apresentado um exemplo simplista da representação destes tipos na linguagem de domínio da aplicação:

```
{
  boleano: true,
```

```
  inteiro: 14,
  número: 54.832,
  nulo: null,
  string: "exemplo",
  array: [1, "2", 3, "quatro", 5],
  objeto: {
    animal: "Baleia"
  }
}
```

Isto significa que é possível gerar qualquer tipo de dados programável em *JSON schemas*. Contudo, o fragmento de código supracitado é determinista, pelo que ainda não é ideal. Aqui entram as **funções de interpolação**, uma ferramenta poderosa da gramática do *DataGen*, que servem para trazer aleatoriedade à geração de valores, restringindo o seu tipo e intervalo de valores possível. A aplicação possui uma vasta documentação destas funções e o seu modo de funcionamento, listando todos os tipos de dados que é possível gerar e as restrições aplicáveis a cada um. Reescrevendo o exemplo acima com funções de interpolação, obtemos o seguinte modelo, que pode produzir uma enorme combinação de resultados diferentes:

```
{
  boleano: '{{boolean()}}',
  inteiro: '{{integer(0, 1000)}}',
  número: '{{float(-100, 500.3)}}',
  nulo: null,
  string: '{{lorem("words", 5, 10)}}',
  array: [1, "2", 3, "quatro", 5],
  objeto: {
    animal: '{{animal()}}'
  }
}
```

Assim, é possível concluir que o *DataGen* permite não só gerar todos os tipos de dados primitivos do formato *JSON*, como randomizar os valores gerados, permitindo decidir o valor final em tempo de execução e não *a priori*.

XML Schemas

O formato *XML* assume uma estrutura hierárquica, onde elementos *XML* são aninhados uns dentro dos outros, à semelhança de um objeto *JSON*. Cada elemento pode possuir atributos

e um valor textual/outros elementos aninhados dentro dele, funcionando como uma espécie de nodo numa árvore de informação.

Um elemento com conteúdo de tipo complexo corresponde ao aninhamento de vários elementos *XML* dentro do original. Como tal, a noção de *array* não existe neste formato: para traduzir essa estrutura de *JSON* para *XML*, é necessário chegar a um compromisso. A abordagem adotada pelo *DataGen* consiste em converter cada elemento do *array* num par chave-atributo, onde a chave é um rótulo do índice.

Sendo assim, chegamos à conclusão de que os tipos de dados primitivos de *XML schemas* são todos simples (não compostos). Contudo, existe uma variedade muito elevada destes, em contraste com os 5 tipos de valores elementares distinguidos em *JSON*, como se pode observar de seguida:

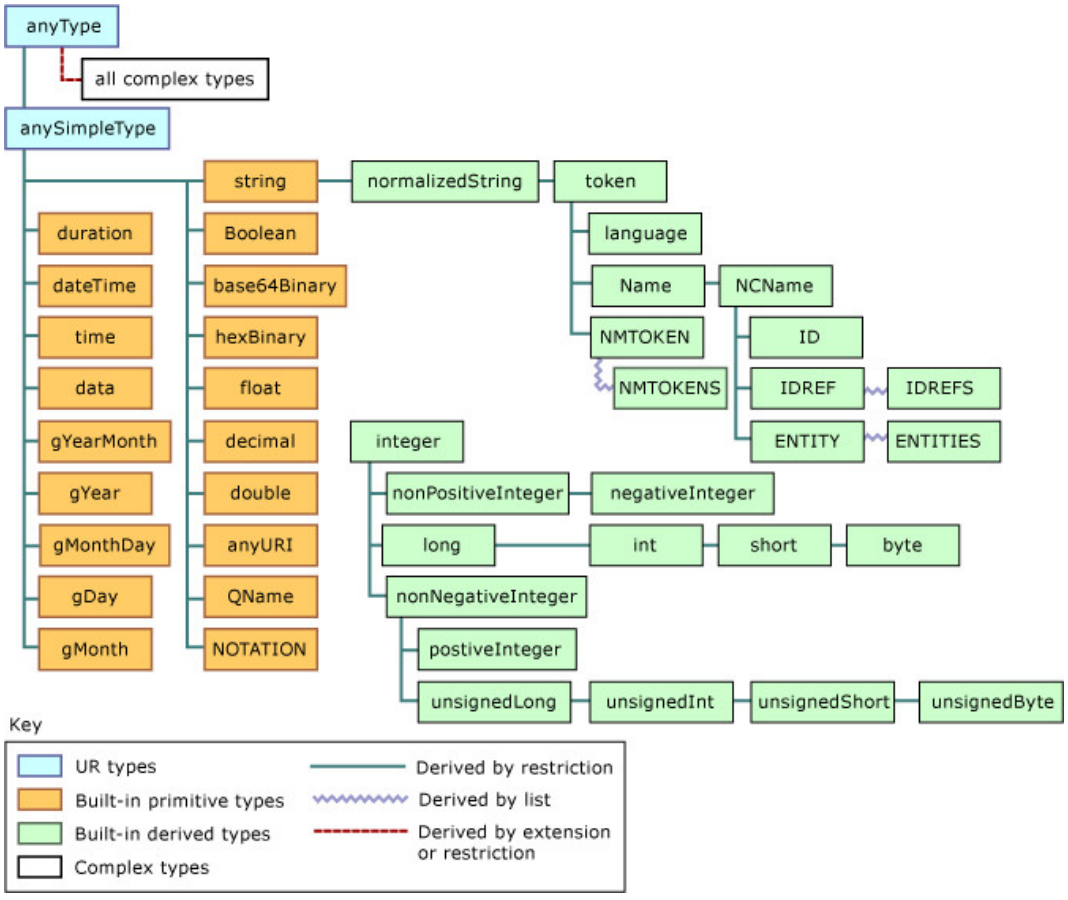


Figura 4: Tipos de dados primitivos do formato *XML*

Após analisar atentamente o gráfico apresentado acima, é possível concluir que todos estes tipos se podem dividir por várias classes:

- **boleanos**: *Boolean*;
- **inteiros**: *integer* e todos os seus tipos derivados;

- **números**: *decimal*, *double* e *float* (os inteiros também estão contidos nesta classe);
- **strings**: *string*, *normalizedString* e todos os seus tipos derivados, *QName* e *NOTATION* (estes dois últimos têm significado na *schema*, mas representam-se por *strings*);
- **binários**: *base64Binary* e *hexBinary*;
- **temporais**: *duration*, *dateTime*, *time*, *date*, *gYearMonth*, *gYear*, *gMonthDay*, *gDay* e *gMonth*;
- **URI**: *anyURI*.

Tendo em conta que os tipos binários, temporais e de URI são representados também por *strings* formatadas, apesar de o seu conteúdo ter um significado específico que se prende ao tipo em si, chega-se então à conclusão de que os tipos primitivos de *XML* constituem basicamente uma ramificação aprofundada dos tipos elementares do formato *JSON*, apenas não existindo noção de *null*.

Assim, é possível afirmar que o *DataGen* possui, teoricamente, a capacidade de gerar qualquer um destes tipos, como foi explicado em 2.2.2. Contudo, é necessário também garantir que possui os meios para restringir o valor gerado à estrutura semântica de cada tipo primitivo, de forma a obedecer à hierarquia apresentada na figura 4: por exemplo, qualquer *token* é uma *normalizedString* válida, mas o oposto é falso. Além do mais, convém também ser possível introduzir aleatoriedade controlada na geração de qualquer um destes tipos. Passando então a analisar caso a caso, os seguintes tipos têm tradução direta para a *DSL* através das funções de interpolação:

- boleanos - função `boolean()`;
- numéricos - funções `integer()` e `float()`, definindo intervalos de valores correspondentes a cada tipo nos argumentos. Por exemplo, o tipo `byte` traduz-se em `integer(-128, 127)` e o `unsignedShort` em `integer(0, 65535)`.
- *date* - função `date()` com indicação do formato de data correspondente ao espaço lexical do tipo, `YYYY-MM-DD` (por exemplo, `date("01-06-1950","20-12-2010","YYYY-MM-DD")`);
- *dateTime* - função `date()` sem argumento para formatação da data, que produz uma data em formato *raw*;
- *time* - função `time()` com argumentos que levem o valor gerado a pertencer ao espaço lexical do tipo, por exemplo `time("hh:mm:ss", 24, false, "12:00:00", "20:30:00")`;
- *gYear* - função `formattedInteger()`, cujo terceiro argumento garante que o inteiro produzido tem sempre o número de algarismos indicado (neste caso, pretende-se 4);
- *gMonth* - interpolação de dois apóstrofes com a função `formattedInteger()`, para gerar valores do género "–06";

- *gDay* - interpolação de três apóstrofes com a função `formattedInteger()`, para gerar valores do género "—15";
- *language* - função `random()`, fornecendo uma lista das abreviações de linguagens mais frequentemente utilizadas.

Todos os tipos primitivos que sobram são *strings* com regras lexicais bastante específicas. Para esse efeito, usar-se-á outra ferramenta do *DataGen*: **funções *JavaScript*** - servem para definir o valor de uma propriedade como o resultado de uma função, onde se pode escrever qualquer código pretendido, que é posteriormente resolvido em tempo de execução. Desta forma, pode-se traduzir as regras do espaço lexical de cada tipo num conjunto de condições lógicas e criar fragmentos de código que giram valores aleatórios dos mesmos.

Assim, pode-se concluir que é também possível gerar qualquer tipo de dados primitivos de *XML schemas* através do *DataGen*, pelo que cumpre o requisito 3.

2.2.3 *Geração de objetos recursivos*

O *DataGen From Schemas* não poderá permitir recursividade infinita, uma vez que necessita de uma condição de paragem para o processo de geração, caso contrário ficaria preso em ciclo infinito. Como tal, será necessário o programa implementar uma restrição de profundidade máxima sobre as definições recursivas em *schemas*, escolha essa que poderá ser dada a tomar ao utilizador, na *frontend*. Haverá um teto predefinido para os níveis de recursividade, que o utilizador poderá alterar conforme os seus *use cases*, num segmento de configurações, gerindo o equilíbrio entre a profundidade que pretende e a complexidade/tempo de geração do *dataset*.

Desta forma, a profundidade máxima dos *datasets* será decidida *a priori*. Tendo em conta que a formatação *JSON* da linguagem de domínio do *DataGen* permite especificar qualquer nível de aninhamento de estruturas, assegura-se então a habilidade de gerar objetos recursivos, dentro de condições razoáveis e imprescindíveis (requisito 4).

2.2.4 *Geração difusa*

A *DSL* do *DataGen* disponibiliza várias ferramentas de geração difusa, que permitem restringir a existência de propriedades do modelo com base em condições lógicas ou probabilísticas (Santos et al., 2021). Estas funcionalidades serão necessárias para lidar com casos mais específicos das definições estruturais, dos quais serão apresentados alguns exemplos de seguida, pertinentes a *XML schemas*:

Elementos nillable

O atributo `nillable` é um atributo de elementos definidos na *schema*, que indica se pode ser atribuído um valor explícito `nil` ao elemento em questão, ignorando o seu conteúdo e permitindo gerar instâncias com o atributo `nil` definido como verdadeiro:

```
% Elemento nillable da schema
<xs:element name="myInt" type="xs:int" nillable="true"/>

% Possíveis elementos XML resultantes
<myInt>5</myInt>
<myInt nil="true"></myInt>
```

Para decidir o conteúdo de cada instância de um elemento `nillable`, usar-se-á as diretivas **missing**/**having** do *DataGen*, que permitem associar a existência de um dado elemento ou propriedade a uma probabilidade arbitrária, de forma a possibilitar que algumas instâncias tenham o conteúdo regular e outras `nil`.

Elementos choice

Um elemento `choice` permite um e um só dos seus elementos *XSD* aninhados no ficheiro final, ou seja, representa um conjunto de elementos mutuamente exclusivos. O *DataGen* disponibiliza a diretiva `or`, que desempenha exatamente o mesmo papel, escolhendo aleatoriamente uma das propriedades especificadas aquando da geração do *dataset*. Logo, existe tradução direta entre estas duas funcionalidades, o que permitirá selecionar o elemento do conjunto em tempo de execução e produzir resultados heterogéneos a partir do mesmo modelo da *DSL*.

Elementos all

Um elemento `all` permite definir um conjunto de elementos que podem aparecer por qualquer ordem. Para este efeito, recorrer-se-á à ferramenta `at_least` do *DataGen*, que permite definir um conjunto de propriedades e o tamanho mínimo do subconjunto delas que será selecionado para o *dataset* final. Um pormenor fulcral desta funcionalidade diz respeito ao facto de as propriedades serem escolhidas aleatoriamente e colocadas no *dataset* final pela ordem que são selecionadas. Pode então tirar-se proveito disto e traduzir um elemento `all` da *schema* para uma diretiva `at_least` que selecione todos os elementos do conjunto, no modelo da linguagem de domínio, efetivamente randomizando a sua ordem.

2.3 TRABALHO RELACIONADO

Nesta secção, será realizada a análise de vários casos de estudo reais com a mesma finalidade que o *DataGen From Schemas*, procurando apurar o quão versáteis e completas são as aplicações existentes presentemente. O foco estará na geração de dados coesos do ponto de vista semântico, visto que ambas as linguagens de especificação possuem uma elevada complexidade, e tendo em conta que o *DataGen* já garante um processo de geração de dados eficiente e escalável. Numa primeira parte, serão focadas as ferramentas de geração em *JSON* a partir de *JSON schemas*. De seguida, serão analisados os geradores análogos para *XML*.

2.3.1 *Geradores de JSON*

O *JSON Schema* é um conceito bastante mais recente que o *XML Schema*, que foi publicado originalmente em 2001. Como tal, as ferramentas de geração a partir de *schemas* de *JSON* que existem não só são em menor quantidade, mas também de menor qualidade, de modo geral, havendo também relativamente pouca documentação das funcionalidades existentes. Nesta subsecção, analisar-se-ão duas aplicações diferentes deste estilo.

JSON Schema Validator and Generator - ExtendsClass

Este programa é extremamente básico e praticamente não tem utilidade no espetro da geração de dados. Demonstra uma enorme falta de funcionalidades, não implementando mecanismos de geração para uma grande percentagem da sintaxe de *JSON schemas*, e uma implementação muito limitada e redutora das funcionalidades que, de facto, possui:

1. a aplicação possui duas grandes facetas: um gerador de *JSON* a partir de *JSON Schema* (e vice-versa) e um validador de *JSON* perante a *schema* introduzida. Ironicamente, estas duas ferramentas que se deviam complementar no fluxo de trabalho do programa são capazes de se contrariar, por vezes, quando o validador verifica que os dados criados pelo gerador a partir de uma *schema* não são válidos em relação à mesma, o que evidencia logo a natureza pobre deste programa de geração;

2. a geração de valores não é aleatória, todo o conteúdo *JSON* que é possível gerar resume-se a um conjunto extremamente básico de valores predefinidos: number: 0.0, integer: 0, *string*: "", boolean: true, object: {} e *array*: []. Este esforço mínimo de geração implica inúmeros outros erros graves no processo de criação de dados, como se irá explicar de seguida;

3. não é capaz de gerar conteúdo algum quando a chave `type` utiliza a notação de *array* para restringir uma instância a um ou mais tipos primitivos, por exemplo `{"type": "number"}` ou `{"type": ["number", "string"]}`;

4. o processo de geração de enumerações é determinista, escolhendo sempre o último valor da enumeração para o *dataset* final. Além disso, é ainda defeituoso, sendo que transcreve mal o valor em questão para o documento *JSON*, na grande maioria dos casos: se for uma *string*, não lhe coloca aspas envolventes e gera um valor inválido; se for um *array*, não lhe coloca os parênteses retos envolventes; se for um objeto, produz `[object Object]` nos dados finais, o que significa que o programa tentou escrever um objeto *JavaScript* diretamente para uma *string* sem o converter primeiro com a função `stringify`;

5. na geração de *strings*:
   - ignora chaves relativas ao comprimento da *string*: `minLength` e `maxLength`. Como o valor gerado para qualquer *string* é predefinido a uma *string* vazia, qualquer restrição com `minLength > 0` implica que o conteúdo *JSON* criado não é semanticamente válido;
   - não é capaz de gerar os formatos embutidos da sintaxe: `date-time`, `time`, `email`, `hostname`, etc.

6. na geração de números, não respeita nenhuma das chaves que é possível especificar relativas ao valor a produzir, nomeadamente `multipleOf` (define um valor do qual o valor gerado deve ser múltiplo) e chaves de alcance: `minimum`, `maximum`, `exclusiveMinimum` e `exclusiveMaximum`;

7. na geração de *arrays*:
   - não é capaz de gerar *arrays* com validação de tuplos, que serve para especificar a ordem dos itens e a *schema* particular de cada um. Independentemente de quantos itens se definam semanticamente na *schema*, o programa produz sempre um simples *array* vazio, o que nem é válido neste contexto;
   - ignora a chave `additionalItems`, que indica se a estrutura pode permitir mais itens do que os especificados ou não. Logo, esta chave é efetivamente fútil da perspetiva deste gerador;
   - da mesma maneira, ignora todas as restantes chaves de especificação semântica de *arrays*: `contains`, `minContains`, `maxContains`, `uniqueItems` e chaves de comprimento (`minItems`, `maxItems`).

8. na geração de objetos:

- nunca gera propriedades adicionais, mesmo tendo a chave `additionalProperties` verdadeira, o que torna a utilização dessa chave inútil do ponto de vista do gerador;
- mesmo que se defina um subconjunto das propriedades como obrigatórias com a chave `required`, no objeto final são sempre produzidas ou todas as propriedades, ou apenas as requiridas (o utilizador pode escolher numa *checkbox* na interface). Não determina a existência das propriedades opcionais de forma aleatória, por exemplo probabilisticamente, o que torna o resultado bastante previsível e desinteressante;
- ignora a chave `propertyNames`, que permite definir um padrão textual ao qual devem obedecer os nomes de todas as propriedades do objeto, gerando propriedades mesmo que não estejam de acordo com o padrão definido;
- ignora as chaves semânticas de tamanho do objeto: `minProperties` e `maxProperties`.

Tendo em conta que a geração dos tipos de dados primitivos desta aplicação da *Extends-Class* é extremamente limitada e bastante defeituosa, como foi exposto acima, não se justifica sequer a testagem de funcionalidades mais complexas da sintaxe de *JSON Schema*, uma vez que não possuem uma base sólida para funcionarem devidamente. Esta ferramenta é bastante rudimentar e a sua implementação da sintaxe deixa muito a desejar, pelo que exibe muita pouca utilidade para casos de aplicação de geração de dados a partir de *schemas*.

JSON Schema to JSON Converter - Liquid Studio

Este conversor é uma aplicação disponibilizada pelo *Liquid Studio* que, embora o seu foco seja o *XML*, também possui algumas ferramentas com operabilidade em *JSON*. Apesar de existir um ambiente de desenvolvimento integrado que centraliza as ferramentas de geração deste grupo, como será novamente mencionado em 2.3.2, este conversor de *JSON Schema* para *JSON* ainda não se encontra incorporado, pelo que a única versão disponível é um programa *online* grátis. Como tal, foi esta a versão utilizada para efeitos de testagem, cujos resultados são apresentados de seguida:

1. introduz aleatoriedade na geração de valores dos tipos primitivos elementares: tipos numéricos, boleanos e *strings*. Os objetos e *arrays* criados são sempre vazios, a menos que se estipulem mais regras semânticas para o seu conteúdo na *schema*. Contudo, ao gerar um *dataset* para uma *schema*, a aplicação guarda o resultado e reaproveita-o, se o utilizador tentar gerar de novo a partir do mesmo modelo. Para obter dados diferentes, é preciso refrescar a página, o que torna a operação mais demorada e piora a experiência do utilizador;

2. resolve o problema do programa anterior relativo ao processamento da chave `type` para a restrição composta de uma instância, até escolhendo aleatoriamente o tipo do valor a gerar, caso seja declarada mais do que uma alternativa;

3. a geração de enumerações também é defeituosa: tanto é capaz de selecionar aleatoriamente um dos valores especificados na chave `enum`, como de gerar um valor aleatório que não está mencionado em lado nenhum;

4. relativamente à geração de *strings*, respeita as chaves de comprimento `minLength` e `maxLength`, gerando *strings* com comprimento dentro dos limites estabelecidos. Contudo, também não suporta a geração de formatos embutidos da sintaxe, o que representa uma grande menos-valia para os casos de aplicação do programa e para a facilidade de instanciação de *strings* formatadas;

5. no que toca à geração de números, não exibe nenhumas melhorias em relação ao gerador da *ExtendsClass* - as chaves `multipleOf` e de alcance são também aqui ignoradas, sendo gerados valores fora dos intervalos permitidos;

6. o processo de geração de *arrays* pouco tem a acrescentar ao da aplicação analisada anteriormente:

   - obecede às chaves `minItems` e `maxItems`, gerando estruturas com um comprimento dentro do intervalo estabelecido. Contudo, esta funcionalidade é pouco flexível, uma vez que os valores gerados são todos aleatórios, inclusivé os seus tipos. O programa não é capaz de conjugar as restrições de comprimento com outras restrições semânticas dos itens, especificadas, por exemplo, através das chaves `contains` ou `prefixItems`;
   - aparenta respeitar também a chave `uniqueItems`, uma vez que todos os itens gerados são aleatórios e, em todos os testes efetuados, nunca coincidiram;
   - contudo, continua também neste programa a faltar uma grande parte das funcionalidades relativas a *arrays* da sintaxe de *JSON Schema*, nomeadamente a validação de tuplos, a geração aleatória de itens adicionais (`additionalItems` a verdadeiro) e as chaves de inclusão `contains`, `minContains` e `maxContains`, que são apenas ignoradas na versão atual.

7. na geração de objetos:

   - obecede à definição de propriedades padrão (`patternProperties`), algo que a aplicação anterior não fazia: esta chave semântica serve para corresponder propriedades cujo nome siga um determinado padrão a uma dada *schema*. Por exemplo, se uma propriedade for definida com o tipo *string* e existir uma regra de padrão

que a inclua e mapeie o valor ao tipo *number*, o gerador reconhece a inconsistência e lança um erro;

- também ignora a chave `additionalProperties` e nunca gera propriedades adicionais, tornando esta ferramenta redundante na *schema*, no contexto de geração de dados;
- gera sempre todas as propriedades definidas, sejam ou não requiridas com a chave `required` - neste aspeto, até apresenta uma desvantagem em relação ao programa anterior, que permitia ao utilizador predefinir se seriam geradas todas as propriedades opcionais, ou nenhuma;
- ignora a chave `propertyNames`, gerando propriedades mesmo que o seu nome não obedeça ao padrão indicado, bem como as chaves de tamanho do objeto `minProperties` e `maxProperties`, falhando em corrigir estas lacunas já existentes no outro gerador analisado.

É possível concluir que, apesar de possuir algumas melhorias em relação à aplicação da *ExtendsClass*, que por si já tornam esta ferramenta mais interessante e versátil, continua a falhar em muitos pontos importantes da geração dos tipos primitivos de *JSON Schema*, que se comprometem não só uns aos outros, mas também a outras funcionalidades mais complexas assentes sobre os mesmos nesta sintaxe. Como tal, pretende-se abordar os problemas expostos na análise realizada a estes programas no *DataGen From Schemas*, criando um produto mais consistente, poderoso e flexível.

2.3.2 *Geradores de XML*

Nesta subsecção, serão exploradas as vantagens e desvantagens de duas aplicações diferentes cujo intuito é gerar dados em *XML* a partir de um modelo *XSD* introduzido pelo utilizador, discutivelmente as mais populares atualmente.

XSD2XML: Online XSD to XML generator

Esta ferramenta provou ser bastante desapontante, exibindo a falta de inúmeras funcionalidades importantes para o processo de geração e uma implementação fraca e bastante limitada das funcionalidades existentes. De seguida, serão expostas as características negativas desta aplicação *web* grátis, apuradas através da sua testagem extensiva:

1. realiza validação sintática das *schemas*, para verificar se estão bem escritas, e ainda algum nível de validação semântica, mas é pouco flexível - por exemplo, no caso de um modelo com recursividade, lança um erro avisando que a *schema* é inválida, embora

não o seja. Uma possível solução para este problema seria ter um limite imbutido de recursividade no seu processo de geração;

2. não reconhece todos os tipos-base *XSD*: não suporta nenhum tipo derivado de `normalizedString` (`token`, `language`, `Name`, `ID`, ...). Alguns destes tipos são raramente usados, mas a ausência dos tipos `ID` e `IDREF`, por exemplo, é bastante significativa, uma vez que exclui automaticamente a possibilidade de criação de *datasets* com identificação e referência de elementos;

3. a geração de valores não é aleatória, o conteúdo *XML* é gerado de acordo com valores predefinidos para cada tipo: *string* -> 'str1234', *int* -> 123, etc;

4. suporte limitado para restrições de tipos - esta ferramenta não é fiável no que toca à geração de tipos restringidos, sendo que o conteúdo gerado raramente respeita as regras estabelecidas no modelo;

5. as listas geradas são inválidas - como foi aprovado e publicado pelo *World Wide Web Consortium* (*W3C*), qualquer tipo de dados de lista deve ter os seus itens separados por *white space*. Contudo, os valores das listas criadas por este programa são todos adjacentes e não há nenhum delimitador que os separe, além de serem todos predefinidos;

6. não suporta a geração de `unions`;

7. não randomiza a ordem do conteúdo de um elemento `all`, simplesmente gera os elementos pela ordem que aparecem no modelo;

8. ignora completamente atributos de ocorrência - gera todos os elementos uma única vez. Isto é especialmente crucial no que toca, por exemplo, a elementos `sequence` - gera sequências de forma elementar, com uma única ocorrência de cada elemento, mesmo que se especifique que o número mínimo de ocorrências da sequência deve ser superior a isso;

9. não é capaz de gerar `choices` - apesar de aparentemente até dar ao utilizador a escolha entre gerar apenas o primeiro elemento de uma `choice` ou todos eles, o autor desta dissertação não conseguiu gerar nem um nem outro: o elemento com a `choice` vinha simplesmente vazio e nenhum dos elementos contidos era gerado;

10. quando o modelo tem conteúdo semi-estruturado, não gera nenhum texto circundante do conteúdo, apenas os elementos. Embora isto seja tecnicamente válido de acordo com o modelo, é muito redutor e pouco representativo dos resultados possíveis. Além disso, e indo de encontro ao ponto 4, se o conteúdo do tipo for esvaziável e o tipo for

restringido de forma a não ter conteúdo, apenas texto circundante, os dados gerados não são válidos;

11. ignora completamente o atributo `nillable`, mesmo que esteja a verdadeiro - nunca gera o elemento respetivo com um atributo `nil`, apenas com o seu valor normal;

12. não suporta inclusão de outras *schemas*, uma vez que não há nenhuma funcionalidade de *upload* de ficheiros;

13. permite predefinir valores personalizados para cada um dos tipos suportados. Contudo, esta funcionalidade é redundante e surge unicamente da incapacidade da aplicação de processar restrições de tipos, visto que esta ferramenta tem exatamente o mesmo propósito que o elemento de restrição `enumeration`.

Após a verificação de todas estas falhas na aplicação, considerou-se que não se justificaria continuar a sua testagem, visto que pouco ou nenhum valor é possível extrair da mesma. O *XSD2XML* é uma ferramenta de geração extremamente básica e simplista, capaz de cumprir à risca apenas os casos mais elementares de *schemas*. Não implementa extensas e relevantes partes da sintaxe *XSD*, como a restrição de tipos simples, e não introduz qualquer tipo de aleatoriedade no processo de geração de dados, sendo os resultados de todas as sub-operações de geração predefinidos e limitados. Além disso, também não é capaz de gerar dados em grandes quantidades, parecendo criar sempre o mínimo de dados possível necessário, o que resulta em *datasets* diminutos, desinteressantes e de pouca utilidade.

XSD to XML Converter - Liquid Studio

O *Liquid Studio* proporciona um conjunto de ferramentas avançado para desenvolvimento em *XML* e *JSON*, com métodos de mapeamento e transformação de dados. Entre as suas várias ferramentas, possui um gerador de dados em *XML* a partir de *schemas* respetivas. Embora este *software* seja comercializado e seja necessário a sua compra para usufruir livremente dele, é possível experimentar o produto completo com um período experimental grátis de 15 dias. O autor da dissertação recorreu a este período experimental para efeitos de testagem, a relatar neste documento, de forma a garantir que os resultados obtidos correspondiam à melhor capacidade do programa.

Este programa revelou-se bastante superior ao *XSD2XML*, corrigindo muitas das suas lacunas e mostrando-se uma ferramenta mais robusta e versátil:

- implementa toda a sintaxe de *XML Schema*, o que significa que suporta todos os tipos-base simples (incluindo os derivados de `normalizedString`), bem como todos os tipos de elementos *XSD*, nomeadamente `choices` e `unions`, que o programa anterior era incapaz de processar;

- introduz um certo grau de aleatoriedade na geração de conteúdo - sempre que gera um novo documento *XML* a partir de uma *schema*:
  - gera valores de determinados tipos simples aleatoriamente, em tempo de execução, nomeadamente tipos numéricos;
  - quando algum elemento tem um número arbitrário de ocorrências (por exemplo, `minOccurs="3" maxOccurs="10"`), o número de vezes que ocorre no *dataset* a ser produzido é determinado durante o processo de geração, de modo aleatório, dentro dos limites impostos.
- incorpora um sistema de inclusão de *schemas*: ao introduzir uma *schema* com um `include`, o programa automaticamente procura o ficheiro referido na mesma diretoria e, caso exista, importa-o também para o ambiente do editor de texto e tem em conta o seu conteúdo na geração de dados, caso contrário avisa o utilizador de que não é capaz de encontrar o ficheiro necessário.

Além disso, este conversor do *Liquid Studio* possui ainda uma outra funcionalidade bastante útil, que não foi mencionada na análise do *XSD2XML*: fornece um conjunto de opções personalizáveis ao utilizador, onde se pode customizar configurações importantes como os limites inferior e superior de recursividade e o tamanho máximo do ficheiro a partir do qual o programa parará de gerar elementos opcionais. Este menu de opções dá uma maior liberdade ao utilizador e permite-lhe ter um melhor controlo sobre o *dataset* gerado.

Apesar de tudo isto, esta ferramenta possui na mesma algumas falhas relevantes que a impedem de ser tão versátil e poderosa como poderia ser, e que devem vir a ser colmatadas na implementação do *DataGen From Schemas*. Estes defeitos são enumerados de seguida:

1. tal como o *XSD2XML*, efetua validação sintática das *schemas*, de forma a verificar se o seu conteúdo está bem formulado, mas validação semântica apenas parcial, não acusando nenhum erro em determinados modelos inválidos - por exemplo, uma *schema* em que se declare um novo tipo lista cujo tipo-base seja também ele um tipo de lista;

2. nem todo o conteúdo gerado é aleatório - os tipos `string` e derivados têm todos um valor predefinido ("string"), exceto os tipos de identificação (`ID`, `IDREF`, `IDREFS`), o que não contribui para a variedade e flexibilidade dos *datasets* produzidos;

3. os valores gerados para os tipos de referenciação `IDREF(S)` são inválidos - para os documentos *XML* de uma *schema* que possua elementos de tipo `ID` e `IDREF` serem válidos segundo a mesma, é necessário que o conteúdo de todos os elementos do tipo `IDREF` corresponda ao conteúdo de algum elemento do tipo `ID` do ficheiro. Ora, esta ferramenta gera os valores do tipo `ID` como "AAAAA", "AAAAB", "AAAAC", etc, mas os valores do tipo `IDREF` como "ID001", "ID002", "ID003", etc. Como tal, estes

valores nunca vão corresponder, pelo que o *dataset* produzido será semanticamente inválido;

4. as listas geradas são inválidas, exatamente como acontece com a aplicação analisada anteriormente - não existe nenhum delimitador a separar os itens da lista, pelo que podem passar, no máximo, por uma lista de um só elemento. Contudo, basta haver uma restrição de comprimento (mínimo) superior a 1 que os resultados deixam de ser válidos;

5. relacionado com o ponto anterior, verificou-se que (pelo menos) a geração de listas é inconsistente - com determinadas *schemas*, o programa gera elementos vazios e não produz qualquer tipo de lista;

6. gera sempre o conteúdo de um elemento `all` pela ordem que aparece no modelo, não randomiza a ordem dos elementos - embora válido do ponto de vista da *schema*, é um método desinteressante no que toca à geração de dados;

7. não é capaz de gerir conteúdo semi-estruturado completo, ou seja, gera apenas o(s) elemento(s) aninhado(s) no elemento com conteúdo misto, sem qualquer texto circundante (igual ao ponto 4 do *XSD2XML*);

8. ignora o atributo `nillable` em elementos - mesmo que esteja a verdadeiro, gera o elemento sempre com o valor normal e nunca com o atributo `nil`.

Concluindo, o *XSD to XML Converter* do *Liquid Studio* constitui uma ferramenta já bastante avançada e capaz de lidar com casos de aplicação complexos, contudo não deixa de ter algumas lacunas lamentáveis que limitam a usabilidade da aplicação e comprometem a confiabilidade dos dados gerados. No entanto, é uma boa aplicação para estudar e servir de inspiração ao *DataGen From Schemas*, sendo possível levantar bastantes requisitos funcionais e marcadores de qualidade para este tipo de programa.

As conclusões tiradas a partir da análise desta ferramenta, bem como das anteriores apresentadas no presente capítulo, serão tidas em conta durante o desenvolvimento do *DataGen From Schemas*, tendo por objetivo corrigir todas as falhas observadas do ponto de vista semântico, criando assim uma aplicação mais robusta e confiável, e implementar as funcionalidades mais vantajosas e úteis encontradas, como por exemplo o menu de definições personalizáveis sobre o *dataset* a produzir.

3

ABORDAGEM PROPOSTA

Esta aplicação será, em termos de estrutura, bastante semelhante ao próprio *DataGen*, uma vez que o cerne da interação com o utilizador é exatamente o mesmo que o seu antecessor: o utilizador introduz o texto a partir do qual se pretende gerar dados (neste caso, uma *schema*), o programa processa o ficheiro, gera um *dataset* e devolve-o ao utilizador.

Tal como no *DataGen*, pretende-se evitar a construção de um único servidor monolítico, o que traria implicações em termos de desempenho e de *bottlenecks* no programa. Será adotada uma arquitetura compartimentalizada, que prioritiza a escalabilidade e a disponibilidade da aplicação, separando a *frontend* e a *backend* em servidores distintos. Desta forma, a falha de um componente individual não compromente a funcionalidade da aplicação inteira, o que também facilita e agiliza a realização de rotinas de manutenção.

Esta arquitetura fragmentada permite proporcionar uma experiência mais agradável e consistente ao utilizador e deixa ainda em aberto a eventual aprimoração de qualquer um dos servidores individuais, atribuindo-lhes mais recursos de computação e mecanismos de tolerância a faltas, como *load balancing* e redundância, caso surja a necessidade de escalar a aplicação para lidar com problemas de tráfego e/ou disponibilidade.

3.1 ARQUITETURA

A arquitetura proposta para a aplicação reflete-se no seguinte diagrama e será explicada detalhamente ao longo deste capítulo, procurando justificar as tecnologias escolhidas e a estrutura projetada.

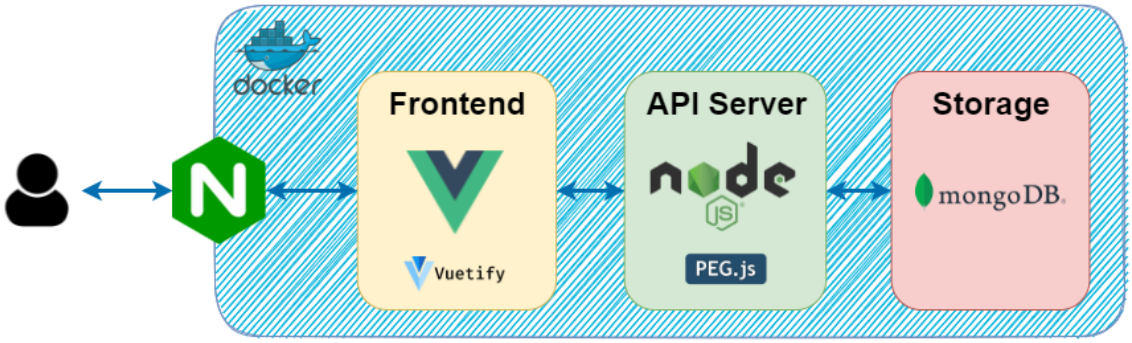


Figura 5: Arquitetura proposta para o *DataGen From Schemas*

Sendo o *DataGen From Schemas* uma aplicação complementar ao *DataGen* com o objetivo de expandir os casos de aplicação do programa original, de forma a incluir a geração de dados a partir de *schemas*, pretende-se manter as funcionalidades já existentes no *DataGen*, nomeadamente a autenticação, a gravação de modelos e a sua disponibilização na plataforma. Não faz sentido guardar também *schemas* na plataforma, uma vez que são formatos estandardizados e reconhecidos globalmente, ao contrário da linguagem específica de domínio, que só é reconhecida no contexto da aplicação.

Tendo em conta que toda a gestão e armazenamento destes dados já está implementada na aplicação original, não se justifica uma nova implementação separada no *DataGen From Schemas*, que seria apenas uma cópia do trabalho já existente. Muito pelo contrário, a hipótese de reutilizar a *backend* e a base de dados do *DataGen* revela-se muito mais interessante e útil, uma vez que permite manter os registos de utilizadores e modelos da aplicação original, assegurando a partilha de todos os dados persistentes entre as duas aplicações, além de poupar muito trabalho repetido e desnecessário. Esta abordagem só traz vantagens também do ponto de vista do utilizador, uma vez que lhe permitirá usar a mesma sessão em ambas as aplicações e editar modelos tanto do *DataGen* como do *DataGen From Schemas* em qualquer um dos dois.

O *deployment* do *DataGen* foi realizado com recurso ao *Docker*, que permite a instanciação do projeto através de *containers* - estruturas independentes do sistema operativo, colocando cada componente da arquitetura num diferente. O *Docker* encapsula a aplicação e é acedido por uma instância de um servidor *NGINX*, que atua como um *proxy* de pedidos para a *frontend* e a *backend*. A base de dados encontra-se escondida do próprio *NGINX*, sendo totalmente invísivel/inacessível a qualquer entidade fora do *Docker*.

Esta estrutura implica mais uma vantagem para a reutilização da *backend* do *DataGen*, uma vez que facilita o acesso à base de dados original, para ter acesso aos mesmos dados, tornando desnecessária qualquer alteração à configuração do servidor *NGINX* que atua sobre a aplicação original.

3.1.1 *Geração de dados server-sided*

O processamento de *schemas* será realizado por compiladores, sendo necessário desenvolver um compilador para *JSON schemas* e outro para *XML schemas*, devido às sintaxes fundamentalmente diferentes que possuem, como foi exposto na secção 2.1. Estes compiladores serão baseados em gramáticas *PEG.js*, uma vez que foi a tecnologia usada para desenvolver o compilador original do *DataGen* e provou cumprir todos os requisitos do programa. Graças a isto, o autor da dissertação já se encontra também bastante ambientado com a ferramenta e conhece bem as suas diversas funcionalidades, pelo que não existirá a típica curva de aprendizagem inicial e será possível proceder para a fase de desenvolvimento mais rapidamente.

Através da análise do ficheiro, os compiladores produzirão uma estrutura de dados intermédia, com todos os dados relevantes extraídos da *schema*. Esta estrutura será então passada a um conversor (também aqui haverá um para cada formato), programa esse que se encarregará de a traduzir para um modelo da linguagem de domínio do *DataGen*, que será depois processado pela aplicação-base a fim de produzir o *dataset* final.

Apesar de o artigo de publicação do *DataGen* (Santos et al., 2021) postular uma abordagem *client-sided*, onde a carga computacional da geração de dados seria colocada inteiramente sobre o *browser* do cliente, esta descrição encontra-se desatualizada e não reflete a estrutura atual da aplicação. Posteriormente, acabou por se mudar o processo de geração de dados para a *backend* do programa, adotando uma abordagem *server-sided*, de forma a expor rotas aplicacionais sem ser necessário clonar o compilador baseado na gramática *PEG.js* e colocar uma cópia em cada servidor da aplicação.

Pela mesma lógica, fará também mais sentido colocar os compiladores e conversores do *DataGen From Schemas* na *backend* da aplicação, de forma a evitar a duplicação destes ficheiros para permitir a criação e exposição de novas rotas aplicacionais, possibilitando assim a utilização deste serviço sem recurso à interface e a sua eventual integração em terceiros. Por um lado, estes programas são independentes do resto da aplicação e desempenham a sua função isoladamente, não precisando de aceder a quaisquer serviços privados da *backend*, nomeadamente a base de dados ou variáveis locais, o que sugere uma abordagem *client-sided* para o processamento das *schemas*. Contudo, o processo global de geração de dados nunca será totalmente *client-sided*, devido à implementação do *DataGen*, pelo que não se justifica esta diferenciação. Além do mais, a carga computacional do processamento das *schemas* e geração dos modelos da *DSL* é pouco significativa, pelo que os ganhos derivados de realizar estas operações no *browser* do cliente (em termos de libertação de recursos no servidor da *backend*, principalmente em termos de *CPU* e memória) são pouco relevantes.

Logo, surge como uma conclusão natural que o *DataGen From Schemas* adote uma abordagem *server-sided* para o processamento das *schemas*, criação dos modelos e consequente geração de dados.

3.2 FRONTEND

O servidor da *frontend* constitui o ponto de entrada da aplicação e é responsável por compilar e exibir a interface da plataforma ao utilizador, gerindo toda e qualquer interação com o mesmo.

De modo geral, pretende-se uma interface bastante próxima da do *DataGen*. Esta aplicação é, no seu âmago, uma extensão do *DataGen* e o seu fluxo de trabalho é análogo - *input* de modelo, processamento e geração, *output* de *dataset* -, diferenciando-se apenas no tipo de *input* (*schemas* vs modelos da *DSL*). Como tal, não há necessidade de "reinventar a roda", podendo-se aproveitar a base já estabelecida pelo *DataGen*, até por uma questão de coesão visual entre as duas ferramentas complementares.

Por este motivo e por outros que serão mencionados de seguida, propõe-se a implementação da interface em *Vue.js*, que também foi a tecnologia usada no *DataGen*. Esta *framework* utiliza uma arquitetura *Model-View-ViewModel* (*MVVM*) que simplifica a programação orientada a eventos utilizada para processar as interações entre o utilizador e a interface.

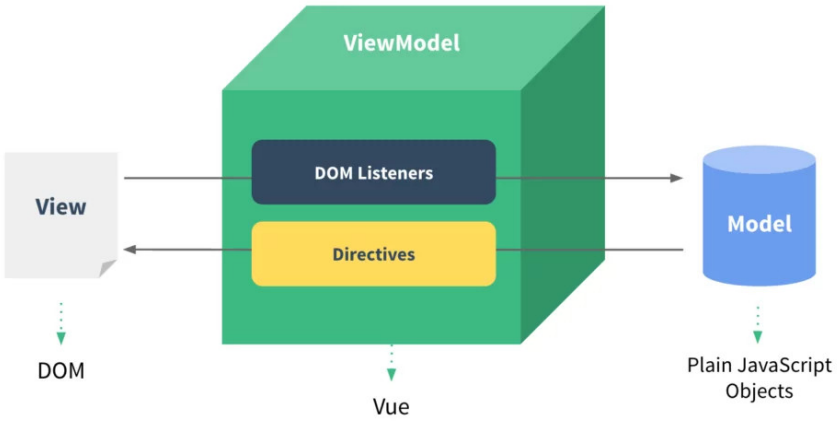


Figura 6: Representação da arquitetura *MVVM*

Com esta arquitetura, o *Vue.js* efetivamente separa a interface do utilizador (*View*) da lógica do programa (*Model*), estabelecendo uma ligação de dados bidirecional reativa entre os dois que contorna a necessidade de recarregar a página quando esta fica desatualizada em relação ao modelo de dados. Assim, torna-se possível traduzir as mudanças no modelo

para o *Document Object Model* (*DOM*) em tempo real, atualizando apenas os elementos correspondentes da interface, o que resulta numa experiência muito mais *user-friendly*, bem como um desempenho mais eficiente do que algumas das suas alternativas, nomeadamente o *React* e o *Angular*.

Em termos operacionais, a interface disponibilizará as seguintes funcionalidades ao utilizador:

- **Autenticação** - implementada com recurso ao padrão *JSON Web Token* (*JWT*), que permite que ambos os servidores sejam *stateless*, contornando a necessidade de manter sessões com os dados dos utilizadores. Todos os pedidos *HTTP* para a *backend* que acedam a dados sensíveis são assinados com este *token*;
- **Geração e *download* de *datasets*** a partir de *schemas* de *JSON*/*XML* - o utilizador poderá introduzir o modelo de duas maneiras: manualmente, num editor de texto, ou fazendo *upload* do ficheiro para a plataforma;
- ***Feedback* de erros** - a aplicação não só informará o utilizador de alguma incompatibilidade do seu modelo com o processo de geração de dados, como também tratará de validar semanticamente a *schema*, reportando qualquer erro que possa encontrar;
- **Gravação e gestão de modelos da *DSL*** - no *DataGen*, o utilizador era capaz de guardar qualquer modelo que criasse, associando-o à sua conta, podendo posteriormente alterar a sua visibilidade na plataforma e editar o seu conteúdo. Pretende-se manter esta característica no *DataGen From Schemas*, dando a opção de guardar o modelo da *DSL* gerado como passo intermédio.

3.3 BACKEND

O servidor da *backend* é responsável por implementar a lógica de negócio e gerir a camada de dados persistentes da aplicação, neste caso os utilizadores e os modelos da *DSL*. Como já foi discutido acima, esta aplicação usará a mesma *backend* que o *DataGen*, expandindo as suas funcionalidades com o processamento de *schemas* e consequente geração de modelos da *DSL* e com novas rotas aplicacionais. Além disso, o servidor continua encarregue da geração de dados, bem como da autenticação dos utilizadores e da gestão dos respetivos modelos.

O servidor está implementado em *Node.js*, que surge como a tecnologia ideal para este contexto - a sua natureza orientada a eventos, assíncrona e *single-threaded* confere-lhe uma distinta eficácia ao lidar com pedidos que exijam pouca carga de computação, como é o caso. Além disso, apresenta ainda bom desempenho em termos de velocidade, uma comunidade muito ativa (o que se traduz em boa documentação e entreajuda), boa escalabilidade e uma extensa biblioteca de pacotes.

No que toca à persistência de informação, não será necessário alterar a base de dados do *DataGen*, que também será partilhada pela nova aplicação de forma a manter o mesmo registo de utilizadores e a se poder gerir juntamente todos os modelos guardados, tanto de uma plataforma como de outra. A base de dados está implementada através de uma instância de um servidor *MongoDB*, tecnologia que foi escolhida originalmente devido à sua natureza *Not Only SQL* (*NoSQL*) orientada a documentos e altamente compatível com *Node.js*, uma vez que ambos possuem suporte para documentos *JSON*. É dotada de uma escalabilidade e desempenho notáveis, derivados do facto de não agrupar os dados relacionalmente, o que permite consultas rápidas, eficazes e sem conflitos, dado que todos os documentos são independentes.

Os dados persistentes estão organizados em três coleções:

- **utilizadores** - nome, email, password (encriptada) e datas de registo e do acesso mais recente;
- **modelos** - modelo da *DSL*, criador, visibilidade (pública ou privada), título, descrição e data de registo;
- **blacklist** - armazena os *tokens JWT* dos utilizadores quando terminam sessão, juntamente com a sua data de expiração, de maneira a serem automaticamente removidos da base de dados assim que passem de validade, evitando que a sessão do utilizador fique sempre aberta.

Por fim, a *backend* tratará ainda de disponibilizar rotas *REST* para possibilitar a utilização do programa sem recurso à interface gráfica. Desta forma, será também possível incorporar as funcionalidades do *DataGen From Schemas* noutras aplicações, através de pedidos *HTTP*. As rotas planeadas são as seguintes:

- `POST` **/api/datagen_schemas/xml_schema/xml** - produz um *dataset* em *XML* a partir de uma *schema XML*, que recebe no corpo do pedido;
- `POST` **/api/datagen_schemas/xml_schema/json** - produz um *dataset* em *JSON* a partir de uma *schema XML*, que recebe no corpo do pedido;
- `POST` **/api/datagen_schemas/json_schema/xml** - produz um *dataset* em *XML* a partir de uma *schema JSON*, que recebe no corpo do pedido;
- `POST` **/api/datagen_schemas/json_schema/json** - produz um *dataset* em *JSON* a partir de uma *schema JSON*, que recebe no corpo do pedido.

Como é possível observar acima, pretende-se que seja possível gerar dados tanto em *XML* como em *JSON*, a partir de qualquer um dos dois tipos de *schemas*. Para este efeito, será

necessário estabelecer algumas normas no que toca à representação de informação de *XML* em *JSON*, de forma a garantir a portabilidade total de dados entre os formatos, devido a algumas características como as que são mencionadas em 2.2.1. Como tal, propõe-se a seguinte taxonomia:

- quaisquer ocorrências dos caracteres `.` e `-` nos nomes das propriedades serão substituídas pelo caractere `_`;
- propriedades com chave igual ao mesmo nível de profundidade de uma dada estrutura serão diferenciadas através de um contador, sendo que se a chave for única, não será numerada. Por exemplo, se um elemento *XML* possuir três elementos aninhados com o nome "morada", em *JSON* passarão a "morada1", "morada2" e "morada3";
- chaves de atributos serão prefixadas com um sinalizador a indicar que se tratam de atributos: por exemplo, um atributo "peso" em *XML* ficará "attr_peso" em *JSON*;
- quando um elemento tiver atributos e conteúdo textual, o conteúdo será colocado numa propriedade *JSON* com chave `value`;
- quando um elemento for `mixed`, o texto fora dos elementos-filhos será colocado em propriedades numeradas `text1`, `text2`, etc. Caso haja apenas uma instância de texto semi-estruturado, ficará numa propriedade com chave não-numerada `text`.

4

CONCLUSÃO

Este relatório de pré-dissertação serviu o propósito de contextualizar o tema central da tese e do projeto a desenvolver, procurando estabelecer um entendimento claro e detalhado do modo de funcionamento do *DataGen*, das linguagens que se pretende incorporar no programa, *JSON* e *XML schemas*, e dos dados que se tenciona gerar.

O estudo da aplicação-base permitiu prever, até certo ponto, como decorrerá o processo de tradução dos ficheiros de especificação para a sua linguagem de domínio e que entraves surgirão derivados da implementação nativa do *DataGen*, enquanto que a análise de casos de estudo com a mesma finalidade permitiu observar e entender as boas e más práticas neste tipo de programas.

Toda esta fase teórica de estudo e preparação permitiu definir objetivos claros e realistas para a dissertação, bem como projetar uma arquitetura apropriada para a aplicação a desenvolver, escolhendo as tecnologias mais adequadas para o efeito e aproveitando a base sólida já estabelecida pelo *DataGen*. Como tal, é dada por concluída esta fase de planeamento e procede-se prontamente para a fase de desenvolvimento do produto.

BIBLIOGRAFIA

General data protection regulation. In *GDPR*, 2018. URL https://gdpr-info.eu/. Accessed: 2021-04-26.

Jason W. Anderson, K. E. Kennedy, Linh B. Ngo, Andre Luckow, and Amy W. Apon. Synthetic data generation for the internet of things. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 171–176, 2014. doi: 10.1109/BigData.2014.7004228.

Jessamyn Dahmen and Diane Cook. Synsys: A synthetic data generation system for healthcare applications. *Sensors*, 19(5), 2019. ISSN 1424-8220. doi: 10.3390/s19051181. URL https://www.mdpi.com/1424-8220/19/5/1181.

Hadi Keivan Ekbatani, Oriol Pujol, and Santi Segui. Synthetic data generation for deep learning in counting pedestrians. In *ICPRAM*, pages 318–323, 2017.

GAO. Artificial intelligence in health care: Benefits and challenges of machine learning in drug development (staa)-policy briefs & reports-epta network. 2020. URL https://eptanetwork.org/database/policy-briefs-reports/1898-artificial-intelligence-in-health-care-benefits-and-challenges-of-machine-learning-in-drug-development-staa. Accessed: 2021-04-25.

Menno Mostert, Annelien L Bredenoord, Monique Biesaart, and Johannes Delden. Big data in medical research and eu data protection law: Challenges to the consent or anonymise approach. 2016. doi: 10.1038/ejhg.2015.239.

Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 399–410, 2016. doi: 10.1109/DSAA.2016.49.

Donald B. Rubin. Statistical disclosure limitation. page 461–468, 1993.

Filipa Alves dos Santos, Hugo André Coelho Cardoso, João da Cunha e Costa, Válter Ferreira Picas Carvalho, and José Carlos Ramalho. DataGen: JSON/XML Dataset Generator. In Ricardo Queirós, Mário Pinto, Alberto Simões, Filipe Portela, and Maria João Pereira, editors, *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, volume 94 of *Open Access Series in Informatics (OASIcs)*, pages 6:4–6:9, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-202-0. doi: 10.4230/OASIcs.SLATE.2021.6. URL https://drops.dagstuhl.de/opus/volltexte/2021/14423.

Dianna M Smith, Graham P Clarke, and Kirk Harland. Improving the synthetic data generation process in spatial microsimulation models. *Environment and Planning A: Economy and Space*, 41(5):1251–1268, 2009. doi: 10.1068/a4147. URL https://doi.org/10.1068/a4147.

Apostolia Tsirikoglou, Joel Kronander, Magnus Wrenninge, and Jonas Unger. Procedural modeling and physically based rendering for synthetic data generation in automotive applications. *arXiv preprint arXiv:1710.06270*, 2017.