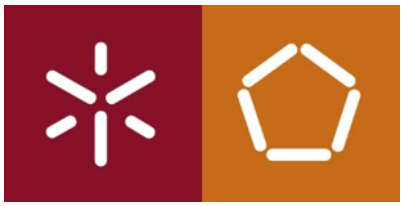

Universidade do Minho

Laboratório em Engenharia Informática

DataGen: Gerador de Datasets em JSON/XML

Relatório de Desenvolvimento

Mestrado Integrado em Engenharia Informática

Equipa:
Filipa Alves dos Santos, a83631
Hugo André Coelho Cardoso, a85006
João da Cunha e Costa, a84775
Válter Ferreira Picas Carvalho, a84464

Coordenador:
José Carlos Ramalho, jcr@di.uminho.pt

27 de Junho de 2021

Resumo

Neste documento, será descrito o processo de implementação da aplicação DataGen.

O DataGen é uma ferramenta poderosa e versátil que permite a prototipagem rápida de datasets e testagem de aplicações de software. Atualmente, há muito poucas soluções que disponibilizam tanto a complexidade como a escalabilidade necessárias para gerar datasets adequados para avaliar o desempenho de uma API de dados ou de uma aplicação complexa, permitindo testar as mesmas com dados heterogéneos e num volume apropriado.

O cerne do DataGen é uma *Domain Specific Language* (DSL) que foi criada para especificar datasets. Esta linguagem passou por vários desenvolvimentos: ferramentas de repetição de propriedades (sem limite), campos difusos (gerados estatisticamente), listas, funções de ordem superior sobre listas, funções de transformação personalizadas, entre outros. O resultado final é uma álgebra diversificada que permite a geração de datasets bastante complexos, lidando com requirimentos muito exigentes e intricados. Ao longo deste relatório, serão dados vários exemplos das possibilidades mencionadas.

Após gerar um dataset, o DataGen dá ao utilizador a possibilidade de gerar uma API de dados RESTful com o mesmo, criando assim um protótipo em execução.

Esta solução já foi testado com casos reais, que serão descritos em maior detalhe ao longo deste documento, nos quais foi capaz de criar os datasets pretendidos com sucesso. Estes casos de estudo permitiram testar o desempenho da aplicação e fazer os ajustes necessários para o seu funcionamento otimal.

O produto vai ser colocado num ambiente de produção e disponibilizado para o público geral, num servidor do Departamento de Informática da Universidade do Minho.

Conteúdo

1 Introdução

Qualquer aplicação e software desenvolvido deve ser minuciosamente testado antes do seu lançamento, de forma a determinar a capacidade do sistema de lidar com quantidades realistas de tráfego e dados, e isso implica a utilização de datasets representativos que se adequam aos seus *use cases*. A criação desses datasets é um processo trabalhoso e prolongado, visto que implica gerar os dados de teste de alguma maneira, num formato compatível com o sistema. Atualmente, não há nenhuma ferramenta para este propósito que seja eficiente, intuitiva e escalável e, como tal, desenvolvedores frequentemente acabam por ter de criar os registos de teste manualmente, o que é incrivelmente ineficiente e demorado, ou aplicando algumas táticas inteligentes a ferramentas existentes para gerir as suas lacunas. Na realidade, muitos projetos não são capazes de progredir para a fase de desenvolvimento devido à falta de dados suficientes e adequados [3].

Mesmo com uma ferramenta de geração fiável, o utilizador pode querer ter controlo sobre os dados dos registos resultantes e ser capaz de poder observar e manipular os mesmos livremente, através de pedidos CRUD, fazendo todas as alterações que achar necessárias. Embora haja produtos capazes de fazer isto - por exemplo o pacote *json-server*, que cria uma API REST falsa completa -, a sua utilização implica que o cliente inicie a aplicação manualmente de cada vez que pretender editar o conteúdo do dataset e que o formato dos dados seja compatível com o software em questão, o que acaba por requerer bastante esforço e configuração extra da parte do utilizador.

Como tal, surgiu a ideia de acoplar o processo de geração e a API REST, de forma a permitir ao utilizador gerar automaticamente dados compatíveis com uma API RESTful integrada - para a qual foi eleito o software Strapi, como será explicado em mais detalhe na secção 3.4 -, permitindo ao utilizador carregar os registos para um servidor API e realizar operações CRUD sobre os mesmos, tanto através da interface como de pedidos HTTP.

Este relatório cobrirá o desenvolvimento da aplicação resultante, DataGen - uma ferramenta mais versátil e poderosa de geração de dados, de acordo com as especificações do utilizador, e processamento subsequente -, procurando explicar as decisões que foram tomadas para a sua arquitetura, bem como todas as funcionalidades implementadas.

2 Trabalho relacionado

Preocupações relacionadas com a privacidade de utilizadores [13] têm sido um tópico de discussão prominente ao longo das últimas décadas, o que eventualmente levou à criação de regulações rigorosas relativas ao tratamento de dados sensíveis, como é exemplo a *General Directive on Data Protection GDPR* da União Europeia [2]. Estas regulações impedem entidades de divulgar informação privada e confidencial, o que, por sua vez, prejudica novas ideias e projetos em crescimento que requiririam acesso a dados semelhantes. De facto, não só a falta de dados disponíveis é um grande problema neste contexto, como também os próprios acordos de partilha de dados, tendo em conta que a sua ratificação tende a demorar, em média, um ano e meio [10], o que prova ser fatal para muitas empresas pequenas e em crescimento.

De forma a contornar estes problemas, organizações têm vindo cada vez mais a adotar o processo de geração de dados sintéticos [16], uma abordagem que foi originalmente sugerida por Rubin em 1993 [1], a fim de criar dados realistas a partir de modelos existentes, sem comprometer a privacidade do utilizador. A perspetiva de ser capaz de criar informação viável que não esteja relacionada a clientes reais é uma solução muito promissora para o dilema da privacidade. Se aperfeiçoada, tem potencial para tornar possível a geração de quantidades de dados consideráveis de acordo com quaisquer *use cases*, evitando a necessidade de aceder a informação de utilizadores reais. Como tal, esta abordagem tem sido incrementalmente adotada e posta ao teste, de maneira a medir a sua eficiência com casos reais [5; 14; 18].

Atualmente, geração de datasets tornou-se um requisito em muitos projetos em desenvolvimento. Contudo, a maioria das ferramentas existentes produzem datasets para contextos específicos, como sistemas de deteção de intrusão em IoT [4], deteção de colheitas e ervas daninhas [6], redes sociais veiculares baseadas em dados de carros flutuantes [12], métricas de canais e contexto 5G [17] e projetos GitHub [9], para citar alguns. Muitas outros existem em enquadramentos diferentes como medicina, bioinformática, previsão metereológica, constância de cor, análise de mercado, etc.

A maior parte das ferramentas investigadas são específicas de domínio. O objetivo da equipa é criar uma ferramenta genérica, mas poderosa o suficiente para gerar datasets para bastantes e variados contextos, com muitos níveis de complexidade. Há alguns softwares disponíveis para estes fins, muitos deles online, mas cobrem maioritariamente a geração de datasets simples e muitas vezes planos (sem aninhamento de dados).

A principal fonte de inspiração para este projeto foi uma aplicação web já existente chamada "JSON Generator", desenvolvida por Vazha Omanashvili [11], visto que foi a ferramenta de geração de dataset mais interessante que a equipa conseguiu encontrar. Conta com uma DSL (*Domain Specific Language*) que é processada e convertida para um dataset em JSON, permitindo ao utilizador gerar um *array* de objetos que seguem uma estrutura predefinida, especificada na DSL.

Em termos de utilidade, é uma ferramenta muito poderosa capaz de gerar estruturas de

dados bastante complexas para qualquer aplicação, com relativamente pouco esforço. Contudo, possui algumas falhas que inicialmente inspiraram a equipa a desenvolver uma solução alternativa que procurar endereça-las.

Especificamente, estas falhas são:

1. Limite de tamanho para a diretiva 'repeat' (total de 100). Naturalmente, o tamanho do dataset produzido é uma das características mais importantes a ter em conta. Para aplicações de maior escala, ter um *array* com poucos elementos não permite fazer testes realistas, uma vez que muito poucas aplicações desenvolvidas em ambiente de produção, se alguma, usam tão poucos registos;

2. Apenas gera JSON. Apesar de ser um dos formatos de ficheiro mais populares para o armazenamento de dados, existem outros que poderiam ser úteis para o utilizador, dado que pode necessitar dos dados num formato diferente para a sua aplicação, por exemplo XML (outro formato de texto aberto [19]), e obviamente seria ideal que não precisasse de os converter manualmente. Isto permitiria uma maior flexibilidade e expandiria os *use cases* aplicáveis à ferramenta;

3. Não gera uma API RESTful para o dataset. Opcionalmente, muitos utilizadores podem querer o seu novo dataset hospedado e exposto por uma API RESTful, possibilitando a sua utilização direta em aplicações web, ou para descarregar uma API personalizada criada especificamente para o seu dataset, para posterior *deployment* numa plataforma à sua escolha;

4. Não tem em conta geração difusa. Alguns elementos do dataset podem ser não-determinísticos e representados por probabilidades. Por exemplo, pode haver um campo que apenas existe se outra propriedade tiver um valor específico - a aplicação devia fornecer alguma maneira de executar cenários destes;

5. Não possui muita variedade e quantidade de dados. Por exemplo, o utilizador pode querer usar nomes de uma lista de celebridades para o seu dataset, visto que permite uma geração mais realista e consistente, o que esta ferramenta não proporciona;

6. Não é multilíngue - os dados usados por esta aplicação estão disponíveis exclusivamente em inglês. Seria mais *user-friendly* dar ao utilizador a escolha da sua linguagem de preferência para o dataset, ao invés de impor uma linguagem predefinida;

7. Não tem em conta integração em aplicações. A geração e download do dataset requere que o utilizador utilize a interface do website, o que não é ideal, visto que muitas aplicações podem querer usar pedidos HTTP para automatizar este processo, para utilização interna;

8. Não suporta ferramentas de programação funcional. Funções como o 'map', 'filter' e 'reduce', que são padrão no paradigma funcional graças à sua simplicidade e eficácia,

não se encontram presentes nesta aplicação. Isto permitiria ao utilizador encadear afirmações e transformar campos para o resultado pretendido de forma homogénea, garantindo à aplicação a capacidade de lidar com *use cases* mais complexos.

Com objetivos claros e uma aplicação inicial para servir de inspiração, a equipa procedeu para a fase de desenvolvimento, começando por decidir a arquitetura do software (isto é, linguagens de programação, frameworks, ferramentas externas, etc), que será explicada nas seguintes secções.

3 Desenvolvimento do DataGen

Contruir a aplicação do zero requiriu uma abordagem "dividir e conquistar", uma vez que ter uma arquitetura de um único servidor monolítico levaria a uma experiência do utilizador menos estável, devido à falta de escalabilidade.

A compartimentalização da aplicação em múltiplos servidores, cada um com a sua função específica, permite uma arquitetura bastante mais sensata e balanceada, visto que proporciona a possibilidade de melhorar individualmente cada um deles. Isto resulta num sistema com muita mais escalabilidade e tolerância a faltas, dado que a falha de um componente não compromete a funcionalidade da aplicação inteira, possibilitando uma rotina de manutenção mais rápida e fácil.

Nas seguintes subsecções, será explicado de que forma a arquitetura corrente foi alcançada e as decisões tecnológicas por detrás de cada componente.

3.1 Arquitetura

A seguinte figura mostra o esquema geral da arquitetura da aplicação, embora simplificada. Esta arquitetura deixa espaço para grandes atualizações futuras, nomeadamente distribuidores de carga tanto na *back-end* como na *front-end*, uma vez que nenhum dos dois mantém um estado (a abordagem JWT permite que os servidores não mantenham sessões com o estado de cada utilizador), e a utilização de MongoDB como uma base de dados distribuída - *sharded cluster*.

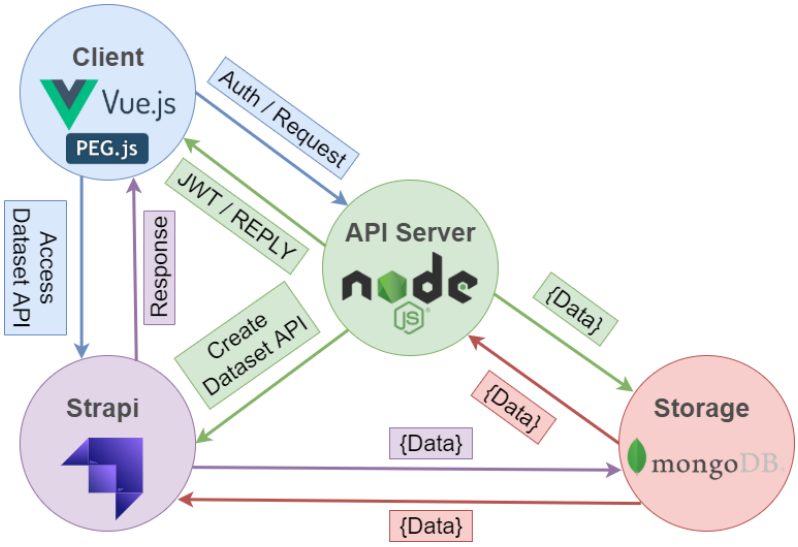

Figura 1: Arquitetura atual do DataGen

3.2 Front-End

O primeiro componente necessário para a aplicação é o servidor de *front-end*, que é responsável por compilar e exibir a interface do website ao utilizador, o que faz dele o ponto de entrada para a aplicação e o seu comportamento programado.

Nesta secção, serão abordadas, de forma geral, todas as funcionalidades implementadas na gramática. Para uma explicação mais exaustiva, é possível consultar a documentação completa no próprio website, que foi desenvolvida com o intuito de guiar o utilizador no uso da DSL. A documentação disponibilizada conta com uma explicação de cada funcionalidade - os seus argumentos e respetivos tipos, o *output* esperado - e exemplos da sua utilização.

3.2.1 Gramática

A aplicação usa um compilador baseado numa gramática PEG.js [8] [15] para processar o *input* do utilizador e gerar o dataset pretendido. A gramática mencionada define uma linguagem específica de domínio (DSL), com sintaxe semelhante a JSON, disponibilizando muitas ferramentas que permitem a geração de datasets complexos e diversificados. Estas ferramentas incluem capacidades relacionais e lógicas, fornecendo meios para os datasets satisfazerem vários tipos de limitações - o que facilita a utilização de frameworks declarativas com esta especificação -, bem como capacidades funcionais, permitindo uma gestão e processamento facilitados de certas propriedades dos datasets.

A primeira e mais fundamental das ferramentas implementadas é a sintaxe semelhante a JSON - o utilizador pode especificar propriedades chave-valor, onde o valor pode tomar qualquer tipo básico ou estrutura de dados JSON, desde inteiros a objetos e *arrays*. O utilizador pode também aninhar estes valores para criar uma estrutura com qualquer profundidade que pretenda.

```
nome: {
  first: ["Hugo", "Cardoso"],
  last: "Miguel"
},
age: 21
```

Para especificar o tamanho do dataset, ou de um *array* aninhado, existe a diretiva **repeat**, onde o utilizador indica a estrutura que pretende replicar (que pode ser qualquer coisa desde um tipo JSON primitivo até um objeto complexo), bem como o número de cópias, ou intervalo de números.

```
nomes: [ 'repeat(150,200)': {
    first: '{{firstName()}}',
    last: '{{surname()}}'
} ]
```

O utilizador pode também definir tantas coleções quantas pretenda num único modelo (são consideradas coleções as propriedades chave-valor no nível superior do modelo) e a aplicação retornará o dataset resultante em sintaxe **json-server** - um objeto com um propriedade por coleção. Durante o processamento do *input*, a aplicação constrói recursivamente tanto

o dataset final como o modelo Strapi para a estrutura especificada, concorrentemente, de maneira a permitir a integração posterior numa API RESTful.

```
{
  nomes: [ 'repeat(10)': '{{fullName()}}' ],
  animais: [ 'repeat(20)': '{{animal()}}' ]
}
```

Para definir o valor de uma propriedade, o utilizador pode também usar interpolação. Para aceder a uma função de interpolação, esta necessita de estar envolta em chavetas duplas. Há dois tipos de funções de interpolação:

- funções que geram valores espontâneos em tempo de execuçãi, de acordo com as instruções do utilizador - por exemplo, existe uma função de geração de números inteiros aleatórios, onde o utilizador precisa de indicar, no mínimo, a gama de valores que pretende para o resultado:

  ```
  id: '{{objectId()}}',
  int: '{{integer(50,100)}}',
  random: '{{random(23, "olá", [1,2,3], true)}}'
  ```

- funções que retornam valores aleatórios de um grupo de datasets incorporados na aplicação por detrás de uma API, onde cada dataset possui informação de uma dada categoria, por exemplo nomes e partidos políticos:

  ```
  nome: '{{fullName()}}',
  partido: '{{political_party()}}'
  ```

Estas funções de interpolação podem também ser interligadas entre si e com strings elementares para gerar strings mais estruturadas, nomeadamente moradas. Algumas destas funções recebem argumentos e, nesse caso, o utilizador tanto pode introduzir manualmente os valores, como pode referenciar outras propriedades definidas acima no modelo, através da variável local **this**, permitindo assim estabelecer relações entre vários campos.

```
freguesia: '{{pt_parish()}}',
distrito: '{{pt_district("parish", this.parish)}}',
morada: 'Rua {{fullName()}}, {{pt_city("district", this.district)}}'
```

Em relação à API de datasets, a equipa realizou uma pesquisa extensiva por datasets com informação útil, utilizou os datasets bem-estruturados que encontrou e aproveitou a informação que pode dos restantes, processando-a para remover erros e normalizar o seu

conteúdo, posteriormente agrupando todos os dados da mesma categoria de forma a criar datasets maiores e dotados de maior complexidade para colocar ao serviço do utilizador.

A equipa também criou alguns datasets originais manualmente, para tópicos considerados relevantes, e introduziu suporte bilíngue - português e inglês - em todos os datasets disponibilizados na aplicação, de forma a dar ao utilizador a liberdade de escolher a linguagem que melhor se adequa ao seu objetivo. Para indicar a sua linguagem de eleição, o modelo do utilizador deve começar com a seguinte sintaxe:

```
<!LANGUAGE pt> ou <!LANGUAGE en>
```

Atualmente, o DataGen tem datasets de suporte para todas as categorias listadas de seguida:

- atores;
- animais;
- capitais;
- centros culturais;
- cidades;
- clubes de futebol;
- continentes;
- desportos;
- dias da semana;
- distritos, cidades, concelhos e freguesias portuguesas;
- empresários portugueses;
- entidades governamentais;
- escritores;
- figuras públicas portuguesas;
- futebolistas;
- hackers;
- marcas;
- marcas de carros;
- meses;
- músicos;
- nacionalidades;
- nomes;

- países;
- palavras-chave;
- partidos políticos;
- profissões;
- políticos portugueses;
- religiões;
- top 100 celebridades;
- top 100 celebridades portuguesas.

A gramática também disponibiliza uma ferramenta chamada **unique()**, à qual o utilizador pode dar uma função de interpolação, ou uma string interpolada com uma função dessas, como argumento. O **unique** garante que as funções de interpolação às quais é aplicado retornam sempre valores únicos, no contexto do dataset final. Isto é especialmente relevante no que toca a funções de interpolação que vão buscar informação aleatória aos datasets de suporte do DataGen, dentro de uma diretiva **repeat**, visto que não há nenhuma garantia base de que retornarão valores sempre diferentes e o utilizador pode não querer que isso aconteça.

Como tal, a função **unique** apenas tem algum efeito quando aplicada em funções de interpolação dos datasets de suporte ou com a função **random** (que pode ser observada num dos exemplos da última página). Desde que seja um destes dois casos (possivelmente interpolados com strings normais) e haja entradas distintas suficientes nos datasets para todos os elementos do **repeat**, o **unique** garante que todos os objetos do dataset resultante terão um valor diferente na propriedade em questão. Se o utilizador usar como argumento uma string com mais do que uma função de interpolação, também não haverá nenhum efeito - poderá haver combinações de valores repetidas no fim.

De seguida, são apresentados dois exemplos: o primeiro demonstra a utilização correta da função **unique**; o segundo mostra casos de uma abordagem errada (não haver valores distintos suficientes para o *repeat*; não ser uma função de interpolação dos datasets de suporte; mais do que uma função de interpolação na string) que ou não funcionarão ou não terão nenhuma garantia de exclusividade mútua para os valores resultantes:

```
[ 'repeat(6)': {
  continente: unique('{{continent()}}'),
  país: unique('Country: {{country()}}'),
  aleatório: unique('{{random(1,2,3,4,5,6)}}')
} ]

[ 'repeat(10)': {
  continente: unique('{{continent()}}'),
  inteiro: unique('{{integer(5,20)}}'),
```

```
  aleatório: unique('{{firstName()}} {{surname()}}')
} ]
```

De volta às propriedades do modelo, o utilizador pode também usar funções de Javascript para definir o seu valor. Existem dois tipos de funções: funções assinadas, onde o nome do método corresponde à chave da propriedade e o resultado do corpo da função ao valor; funções "*arrow*"anónimas, que são usadas para indicar apenas o valor da propriedade (a chave precisa de ser especificada antes da função).

```
name: "André",
email(gen) {
  var i = gen.integer(1,30);
  return `${this.nome}.${gen.surname()}${i}@gmail.com`.toLowerCase();
},
probabilidade: gen => { return Math.random() * 100; }
```

Dentro destas funções, o utilizador é livre de escrever todo o código de Javascript que pretenda, que será posteriormente executado para determinar o valor da propriedade. Desta forma, torna-se possível incorporar algoritmos mais complexos na lógica de construção do dataset, permitindo uma ferramenta de geração mais especializada e versátil. Dado que o utilizador tem acesso a toda a sintaxe de Javascript, pode também fazer uso de operadores relacionais e lógicos para elaborar condições sobre os dados pretendidos, bem como de métodos funcionais (por exemplo "map"e "filter", que são implementados pelo Javascript).

Dentro destes blocos de código, o utilizador tem acesso total a qualquer propriedade declarada acima no modelo da DSL, através da variável local **this**, bem como a qualquer função de interpolação disponível na gramática, através da variável local **gen** - sempre que usar uma função para definir o valor de uma propriedade do modelo, o utilizador necessidade de declarar este argumento na sua assinatura, ao qual pode aceder posteriormente no corpo da função para aceder às ditas funções de interpolação. Tudo isto pode ser observado no exemplo acima.

A gramática também permite geração difusa de propriedades, isto é, a imposição de restrições sobre a existência de certas propriedades, com base em condições lógicas ou probabilidades. A gramática possui quatro ferramentas diferentes para este fim:

- diretivas **missing**/**having** - como argumento, recebem a probabilidade de as propriedades contidas nelas (não) existirem no dataset final; esta probabilidade é calculada para cada elemento, originando assim um dataset onde alguns elementos podem ter as propriedades em questão e outros não:

  ```
  missing(50) { prop1: 1, prop2: 2 },
  having(80) { prop3: 3 }
  ```

- condições **if... else if... else** - estas funcionam como em qualquer linguagem de programação: o utilizador pode usar operadores relacionais e outros condicionais para criar condições e juntá-las com a ajuda de operadores lógicos. O objeto final terá as propriedades especificadas no bloco da primeira condição que se verificar (ou eventualmente nenhuma delas, se todas as condições forem falsas). Nestas condições, semelhante ao modo de funcionamento das funções, o utilizador tem acesso ilimitado a todas as propriedades declaradas acima no modelo da DSL, bem como a todas as funções de interpolação, o que cria a possibilidade de relacionar causalmente diferentes propriedades:

```
tipo: '{{random("A","B","C")}}',
if (this.tipo == "A") { A: "tipo A" }
else if (this.tipo == "B") { B: "tipo B" }
else { C: "tipo C" }
```

- a diretiva **or** - a gramática disponibiliza esta operador lógico para permitir a prototipagem ágil de propriedades mutuamente exclusivas, onde apenas uma delas será selecionada aleatoriamente para cada objeto (note-se que não faz sentido criar uma diretiva **and**, pois isso seria equivalente a simplesmente listar as propriedades pretendidas normalmente no modelo da DSL):

```
or() {
  prop1: 1,
  prop2: 2,
  prop3: 3
}
```

- a diretiva **at_least** - dentro deste bloco, o utilizador escreve um conjunto de propriedades e dá como argumento o número mínimo dessas propriedades que deve estar presente no objeto final. O compilador seleciona um conjunto dessas propriedades aleatoriamente, entre o mínimo dado e o total:

```
at_least(2) {
  prop1: 1,
  prop2: 2,
  prop3: 3
}
```

Finalmente, a gramática também disponibiliza uma implementação das ferramentas de programação funcional fundamentais - **map**, **filter** e **reduce**. O utilizador pode uma ou várias destas funções com um valor *array* (de qualquer uma das várias formas de declaração

de *arrays* disponibilizadas na gramática). Sintaxe *shorthand* não é permitida, pelo que o utilizador deve sempre abrir chavetas para o bloco de código dentro da função. Tirando isso, esta implementação funciona exatamente como a implementação nativa do Javascript: o utilizador pode declarar a função dentro *map/filter/reduce* ou usar sintaxe anónima para as variáveis; nas variáveis, pode declarar apenas o valor atual ou, adicionalmente, qualquer uma das outras variáveis complementares menos comuns. De seguida, é possível observar vários exemplos distintos do uso destas ferramentas:

```
map: range(5).map(value => { return value+1 }),
filter: [0,1,2].filter(function(value, index) {return [0,1,2][index]>0}),
reduce: range(5).reduce((accum, value, index, array) => {
                                          return accum + array[index] }),
combinados: range(5).map((value) => { return value+3 })
                  .filter(x => { return x >=  5})
                  .map(x => { return x*2 }).reduce((a,c) => {return a+c})
```

3.2.2 Estruturas de dados intermédias

Após o processamento do modelo, o compilador gera uma estrutura de dados intermediária com o dataset final, que pode então ser traduzida para JSON ou XML, de acordo com a preferência do utilizador, bem como uma estrutura de dados com o modelo de Strapi correspondente, para possível integração posterior na API RESTful.

Além disso, a gramática encarrega-se também de verificar se existem erros no modelo da DSL durante o seu processamento, criando uma lista com todos os que encontra para apresentar posteriormente ao utilizador. A aplicação é incapaz de gerar o dataset enquanto o modelo possuir erros. Como tal, ao apresentá-los, por cada um, é dada uma mensagem a indicar o problema em questão, bem como a posição exata de onde ocorreu (linha e coluna). Desta forma, o utilizador sabe exatamente o que tem de corrigir e onde, tornando o processo bastante simples e *user-friendly*.

3.2.3 Geração *client-side*

Este projeto foi desenvolvido com a intenção de ser uma aplicação web, mais especificamente um website com características *user-friendly*. Uma abordagem *server-sided* implicaria processar os modelos da DSL no servidor da *back-end*, o que não seria sensato, uma vez que o compilador PEG.js não requere acesso a quaisquer serviços privados (ou seja, bases de dados) escondidos pela própria *back-end*.

Como tal, uma abordagem *client-sided* faz mais sentido para esta aplicação em particular, libertando recursos (principalmente CPU e memória) do servidor de *back-end* e passando a carga de computação do dataset gerado para o *browser* do cliente, usando a *back-end* como um servidor de API.

Existem muitas *frameworks* centradas em *browsers client-sided* e a equipa decidiu que Vue.js seria a alternativa mais adequada para esta aplicação. Permite ligação de dados bidirecional reativa - conexão entre atualizações ao modelo de dados e a interface (UI), o que cria uma experiência mais *user-friendly*, dado que permite exibir as mudanças no DOM em tempo real, em vez de forçar a página a recarregar (como no paradigma *server-sided* tradicional. Outras razões como a flexibilidade (do estilo e comportamento programado dos componentes) e o desempenho (mais eficiente que React e Angular) foram também fatores decisivos na escolha desta *framework* específica.

Após a decisão da *framework*, a equipa começou a desenvolver a interface, que é dotada das seguintes características:

- Autenticação - usa JWT (JSON Web Tokens) que são enviados no cabeçalho 'Authorization' em todos os pedidos HTTP que precisam de ser validados pela *back-end* (que acedam a dados restritos do utilizador);
- Geração e download do dataset e/ou respetiva API - como foi mencionado previamente, a aplicação usa um compilador PEG.js para converter a DSL em JSON ou XML, bem como Strapi para a API (como será abordado na subsecção 3.4);
- Guardar modelos da DSL - utilizadores autenticados têm a opção de guardar os modelos criados e, opcionalmente, disponibilizá-los na plataforma, de forma a que outros utilizadores os possam usar, sendo capazes de realizar operações CRUD nos seus próprios;
- Documentação - tendo em conta que a DSL tem uma estrutura definida, é essencial que o utilizador tenha acesso a estas diretrizes a qualquer momento;
- Descrição da equipa - o utilizador pode querer contactar a equipa diretamente, pelo que existe uma página dedicada aos elementos para encontrar toda esta informação e uma breve descrição da equipa.

A *front-end* precisa de aceder a informação persistente como os dados do utilizador e modelos guardados, que é alcançável através da API RESTful da *back-end* (que será observada em maior detalhe na subsecção seguinte).

3.3 Back-End

A aplicação precisa de um servidor de *back-end* para vários propósitos, nomeadamente armazenamento de dados, autenticação de utilizadores e geração da API para os datasets.

Nenhum dos requisitos mencionados exige poder de computação elevado para cada pedido enviado pelo utilizador, o que foi uma das principais razões pelas quais a equipa optou por um servidor Node.js - é construído de maneira a lidar com qualquer pedido recebido pouco exigente em termos de CPU, graças ao seu modelo IO *single-threaded*, orientado a eventos e não-bloqueante. Uma vez que é também escalável, apresenta bom desempenho em termos de velocidade e possui ainda uma vasta seleção de pacotes (disponíveis no registo **npm**).

Para armazenar todos os dados que precisam de ser consistentes, o servidor da *back-end* acede a uma instância de um servidor MongoDB, que foi escolhido devido à sua escalabilidade (os dados não são agrupados relacionalmente, o que significa que cada documento pode ser encontrado numa instância de nodo diferente sem causar quaisquer conflitos, visto que os nodos são independentes) e à compatibilidade direta que apresenta com o Node.js, dado que ambos aceitam documentos JSON.

Atualmente, a aplicação utiliza três coleções na base de dados MongoDB:

- **utilizadores** - armazena toda a informação relativa aos utilizadores, nomeadamente o seu nome, email, password (encriptada) e datas de registo e do acesso mais recente;
- **modelos** - guarda todos os modelos da DSL e o utilizador a quem pertencem, a sua visibilidade (pública ou privada), título, descrição e data de registo;
- **blacklist** - armazena os tokens JWT dos utilizadores após o seu *log-out* e a sua data de expiração, de maneira a serem automaticamente removidos da base de dados uma vez que expirem.

A autenticação do utilizador permite-lhe aceder aos seus modelos da DSL guardados e realizar operações CRUD sobre os mesmos. Devido à integração do Node.js, foi escolhida uma abordagem JWT (JSON Web Token) para autenticar o utilizador - após a submissão das suas credenciais, o sistema compara-as com as presentes na base de dados e, caso correspondam, é devolvido o token na resposta HTTP, que é necessário para qualquer outro pedido que aceda a informação crítica do próprio utilizador (estes tokens expiram após um determinado período de tempo, por motivos de precaução e segurança).

Após o *log-out*, o mesmo token é adicionado à *blacklist* para fornecer uma camada extra de segurança, impedindo um utilizador que tenha obtido acesso ao JWT de outro (caso este tenha terminado a sessão antes da data de expiração) de submeter pedidos assinados com o mesmo.

A geração da API é um processo mais complexo e tem uma subsecção dedicada apenas a si (3.4), onde são explicadas os passos que foram tomados a fim de obter uma API REST totalmente funcional para qualquer dataset gerado.

3.4 API Strapi

O DataGen também disponibiliza uma outra funcionalidade importante que é a geração da API de dados para o dataset criado. É uma ferramenta útil, uma vez que muitos utilizadores podem querer realizar operações CRUD sobre os dados criados, ou até utilizar a própria API para trabalho futuro.

A ferramenta escolhida para criar este API foi o Strapi [20], um dos melhores sistema de gestão de conteúdo que existe atualmente. O Strapi gera automaticamente uma API REST e permite correr múltiplas APIs simultaneamente. É também bastante simples de configurar e

suporta diferentes sistemas de gestão de base de dados, nomeadamente PostgreSQL, MySQL, SQLite e MongoDB, sendo que a equipa optou por esta última opção para o projeto.

Também foi considerado o JSON-server para este componente do trabalho, mas esta ferramenta é pouco escalável, uma vez que apenas é capaz de correr uma única API de cada vez, o que seria muito pouco eficiente neste contexto. Contudo, o Strapi também apresentou alguns obstáculos, como a estratégia complicada que usa para guardar dados estruturados (por exemplo, um *array*) e a forma de importação de dados, que foram ultrapassados com sucesso.

O processo de construção da API começa dentro da gramática do programa, uma vez que o modelo Strapi é criado de maneira recursiva, concorrentemente com o próprio dataset. Esta estratégia permitiu otimizar bastante a execução do programa e o tempo necessário para gerar o modelo. Por exemplo, sempre que o compilador encontra um *array*, uma vez que o Strapi não possui um tipo próprio para o representar, é criado um componente com os elementos do *array* e é guardada uma referência para esse componente no modelo. Este processo recursivo continua com uma lógica análoga até chegar à raiz do modelo da DSL, que corresponde à coleção como um todo.

Após criar o modelo, estes dados são redirecionados para uma aplicação auxiliar que os processa e converte para o formato Strapi usual. No fim, o programa fica com o modelo acabado e um *array* com todos os componentes criados. A este ponto, o utilizador pode fazer *download* de uma versão do modelo da API em ZIP, se assim pretender, e para facilmente a correr no seu dispositivo pessoal.

Além disso, o DataGen é ainda capaz de popular a API recém-criada com o dataset gerado através de um conjunto de pedidos POST. Devido à falta de métodos de importação de ficheiros inteiros do Strapi, este ciclo de pedidos POST foi a solução encontrada pela equipa para disponibilizar uma API REST temporariamente populada, com todos os métodos HTTP padrão funcionais.

3.5 Interoperabilidade

Note-se que o propósito do DataGen é criar datasets de acordo com as instruções do utilizador, em JSON ou XML. Embora a especificação do modelo possa envolver código de Javascript, sob a forma de funções ou condições, como foi explicado na subsecção anterior, isto não está relacionado com a aplicação cliente de qualquer maneira. O DataGen meramente gera datasets de teste - pode ser usado para qualquer tipo de aplicação que aceite dados em formato JSON/XML, seja a aplicação escrita em Javascript, Python, C++ ou outra linguagem.

3.6 *Deployment*

Com o objetivo de tentar mitigar problemas que pudessem surgir da heterogeneidade de computadores, a aplicação foi *deployed* com recurso do **Docker**, que permite a instanciação

do projeto utilizando *containers* - estruturas independentes do sistema operativo no qual o **Docker** está instalado.

Como é possível observar na figura 1, o programa tem 4 componentes principais: *front-end* em Vue.js, *back-end* em Node.js, Strapi e MongoDB. Esta divisão permite criar um *container* para cada uma destas entidades, uma vez que cada uma tem uma função específica e, caso haja necessidade de comunicação, utilizam REST, como já foi abordado.

Para este propósito, é disponibilizado um ficheiro - **docker-compose.yml** - que contém todas as instruções necessárias para obter o serviço final com os vários *containers* que necessita. Para além deste ficheiro, existem ainda **Dockerfiles** - uma por componente - com os passos necessários para a criação da imagem do respetivo componente (que necessita da sua própria diretoria dentro do projeto), que serão utilizadas para a geração do *container* respetivo.

A execução do comando *docker-compose up* resulta na criação de um serviço constituído pelos quatro *containers* abordados acima. Por defeito, a *front-end* será criada na porta **8080**, que deverá ser exposta a tráfego HTTP para obter a aplicação principal funcional. Os restantes *containers* comunicam através de uma *network* interna gerada pelo **Docker** com a *front-end*, pelo que não deverão ser expostos ao exterior. Através do serviço **NGINX**, que atua como *proxy* de pedidos na *front-end*, o utilizador consegue aceder à *back-end* ou ao **Strapi** sem comprometer a segurança do programa. O *container* com MongoDB, pelo contrário, nunca deverá ser exposto de alguma forma ao exterior, visto que é da responsabilidade da *back-end* aceder à base de dados.

4 Resultados

Uma das prioridades durante o desenvolvimento da aplicação foi a testagem do DataGen com casos reais o mais cedo possível, de maneira não só a validar o conceito e a sua operabilidade, como também a observar que tipo de restrições e pedidos surgiam com maior frequência na criação de datasets de teste, como forma de receber *feedback* externo confiável em relação a ferramentas úteis que devessem ser implementadas.

A equipa estabeleceu contacto com outras entidades e recebeu pedidos para criar datasets de teste para sistema reais, usando o DataGen, que envolviam o uso de lógica de geração bastante complicada com muitas restrições. Estas oportunidades ajudaram a fomentar o crescimento do DataGen, uma vez que surgiram bastantes ideias novas da análise de requisitos dos pedidos, que foram posteriormente implementadas na aplicação. Além disso, provaram ainda a funcionalidade da aplicação e aumentaram a sua credibilidade, dado que os resultados obtidos foram adequados e bastantes positivos.

A fim de manter este relatório conciso, serão mostradas apenas as partes mais importantes de um dos casos de aplicação mencionados mais complexos que o grupo realizou - autos de eliminação [7].

Autos de eliminação são uma estrutura que deve ser criada e cuidadosamente preenchida de maneira a eliminar documentação que atinja o prazo da sua conservação administrativa de forma segura. Este processo é cada vez mais importante hoje em dia, visto que o método padrão de armazenamento de informação já mudou maioritariamente para formato digital e esse processo não é executado da forma correta frequentemente, o que aumenta o risco de perda de informação a longo prazo. De forma a corrigir isto, a eliminação de documentação expirada é tão importante quanto o armazenamento de informação nova.

A geração destes documentos envolve uma lógica complexa, com muitas propriedades que se relacionam diretamente entre si, de acordo com os seus valores, e outros cujo valor deve pertencer a um grupo rígido de possibilidades. Cada auto tem uma fonte de legitimização, cujo tipo pode tomar um de cinco valores de string diferentes. De acordo com o tipo da fonte do auto, os seus fundos (entidades públicas) variam também entre uma única entidade, em alguns casos, e um *array* de várias:

```
fonteLegitimização: {
  tipo: '{{random("PGD/LC", "TS/LC", "PGD", "RADA", "RADA/CLAV")}}',
  fundos(gen) {
    if (["PGD/LC","TS/LC","PGD"].includes(this.fonteLegitimização.tipo))
      return [gen.pt_entity()]
    else {
      var arr = [], i
      for (i=0; i < gen.integer(1,5); i++) arr.push(gen.pt_entity())
      return arr
  }
```

```
}
```

Continuando, cada auto tem um *array* de classes. Caso o tipo da fonte de legitimação seja "PGD/LC"ou "TS/LC", cada classe tem um código; caso contrário, pode ter um código, uma referência ou ambos. O código da classe pode ser composto por 3 ou 4 níveis, cada um com a sua categorização específica:

```
classes: [ 'repeat(2,5)': {
  if (["PGD/LC","TS/LC"].includes(this.fonteLegitimização.tipo)) {
    code(gen) {
      var nivel1 = gen.random(...gen.range(100,950,50))
      var nivel2 = gen.random(10,20,30,40,50)
      var nivel3 = gen.integer(1,999,3)
      var nivel4 = gen.random("01","02")

      var classe = nivel1 + '.' + nivel2 + '.' + nivel3
      if (Math.random() > 0.5) class += '.' + nivel4
      return classe
    }
        }
  else {
    at_least(1) {
      código(gen) { (...) //igual à função acima },
      referência: '{{random(1,2,3,55,56)}}'
    }
  }
} ]
```

Existem também propriedades de anos cujo valor deve pertencer aos últimos 100 anos:

```
anoInicio: '{{integer(1921,2021)}}',
anoFim(gen) {
  var ano = gen.integer(1921,2021)
  while (ano < this.anoInicio) ano = gen.integer(1921,2021)
  return ano
}
```

Finalmente, há dois campos relacionados, o número de agregações e a lista correspondente de agregações, sendo que o tamanho da lista deve corresponder ao número indicado:

```
númeroAgregações: '{{integer(1,50)}}',
aggregations: [ 'repeat(this.númeroAgregações)': {
```

```
    código: '{{pt_entity_abbr()}} - {{integer(1,200)}}',
    título: '{{lorem(3,"words")}}',
    ano: '{{integer(1921,2021)}}',
    if (["PGD/LC","TS/LC"].includes(this.fonteLegitimização.tipo)) {
        naturezaIntervenção: '{{random("PARTICIPANT","OWNER")}}'
    }
} ]
```

5 Conclusão

Ao longo deste relatório, foi discutido o desenvolvimento de um gerador de dados multilíngue, com integração de uma API REST embutida. A intenção por detrás deste projeto era criar uma ferramenta poderosa e versátil que permitisse a prototipagem rápida de datasets para testar aplicações de software, um assunto bastante importante e comum hoje em dia que parece passar surpreendentemente despercebido, apesar da sua vasta relevância.

Sejam projetos em pequena escala de estudantes universitários ou softwares empresariais grandes e complexos, qualquer aplicação deve ser meticulosamente testada ao longo do seu desenvolvimento, o que requere que a equipa de desenvolvedores use dados realistas e em quantidade adequada para popular o sistema. Mesmo hoje, este processo está muito mal otimizado, acabando muitas vezes por levar a uma geração manual de registos, que é muito morosa, ou, ainda pior, a uma testagem escassa e ineficiente do sistema, com poucos dados, possivelmente resultando em conclusões erróneas, defeitos que passam despercebidos e gargalos perigosos na arquitetura do programa.

Como tal, o DataGen surge como uma aplicação simples e rápida de usar que permite ao utilizador prototipar modelos de dados rápida e intuitivamente, de acordo com os seus *use cases*, e colocar o seu programa em prática com um dataset de teste recém-gerado, com valores adequados e realistas. Desta forma, o DataGen consegue automatizar o processo de geração de dados e facilitar o papel do utilizador no mesmo, melhorando a sua experiência e libertando mais tempo e recursos para o projeto em si.

O DataGen foi colocado em prática repetidamente com casos reais e provou ser capaz de criar datasets complexos e de tamanhos consideráveis para aplicações terceiras. O produto será colocado num ambiente de produção e disponibilizado para o público geral, num servidor do Departamento de Informática da Universidade do Minho, após uma fase de desenvolvimento bastante trabalhosa, mas bem-sucedida. A plataforma será *open-source* e os seus conteúdos serão disponibilizados no Github.

Referências

[1] D.b. statistical disclosure limitation. page 461–468, 1993.

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

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

[4] Yahya Al-Hadhrami and Farookh Khadeer Hussain. Real time dataset generation framework for intrusion detection systems in iot. *Future Generation Computer Systems*, 108:414–423, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19322678, doi:https://doi.org/10.1016/j.future.2020.02.051.

[5] Anat Reiner Benaim, Ronit Almog, Yuri Gorelik, Irit Hochberg, Laila Nassar, Tanya Mashiach, Mogher Khamaisi, Yael Lurie, Zaher S Azzam, Johad Khoury, Daniel Kurnik, and Rafael Beyar. Analyzing medical research results based on synthetic data and their relation to real data results: Systematic comparison from five observational studies. 2015. URL: https://unece.org/fileadmin/DAM/stats/documents/ece/ces/ge.46/20150/Paper_33_Session_2_-_Univ._Edinburgh__Nowok_.pdf/, doi:10.2196/18910.

[6] Maurilio Di Cicco, Ciro Potena, Giorgio Grisetti, and Alberto Pretto. Automatic model based dataset generation for fast and accurate crop and weeds detection. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5188–5195, 2017. doi:10.1109/IROS.2017.8206408.

[7] Elimination records. https://clav.dglab.gov.pt/autosEliminacaoInfo/. Accessed: 2020-05-02.

[8] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004. Accessed: 2021-04-20. URL: https://bford.info/pub/lang/peg.pdf.

[9] Georgios Gousios. The ghtorent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 233–236, 2013. doi:10.1109/MSR.2013.6624034.

[10] Bill Howe, Julia Stoyanovich, Haoyue Ping, Bernease Herman, and Matt Gee. Synthetic data for social good. 2017.

[11] JSON Generator. https://next.json-generator.com/4kaddUyG9/. Accessed: 2020-05-04.

[12] Xiangjie Kong, Feng Xia, Zhaolong Ning, Azizur Rahim, Yinqiong Cai, Zhiqiang Gao, and Jianhua Ma. Mobility dataset generation for vehicular social networks based on

floating car data. *IEEE Transactions on Vehicular Technology*, 67(5):3874–3886, 2018. doi:10.1109/TVT.2017.2788441.

[13] 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.

[14] Beata Nowok. Analyzing medical research results based on synthetic data and their relation to real data results: Systematic comparison from five observational studies. 2020. Accessed: 2021-05-03.

[15] PegJS. https://pegjs.org/. Accessed: 2021-04-20.

[16] Haoyue Ping, Julia Stoyanovich, and Bill Howe. Datasynthetizer: Privacy-preserving synthetic datasets. In *Proceedings of SSDBM '17*, 2017. doi:10.1145/3085504.3091117.

[17] Darijo Raca, Dylan Leahy, Cormac J. Sreenan, and Jason J. Quinlan. Beyond throughput, the next generation: A 5g dataset with channel and context metrics. In *Proceedings of the 11th ACM Multimedia Systems Conference*, MMSys '20, page 303–308, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3339825.3394938.

[18] Debbie Rankin, Michaela Black, Raymond Bond, Jonathan Wallace, Maurice Mulvenna, and Gorka Epelde. Reliability of supervised machine learning using synthetic data in health care: Model to preserve privacy for data sharing. 2020. doi:10.2196/18910.

[19] REGULAMENTO NACIONAL DE INTEROPERABILIDADE DIGITAL (RNID). https://dre.pt/application/file/a/114461891. Accessed: 2020-04-21.

[20] Design APIs fast, manage content easily. https://strapi.io/. Accessed: 2020-04-21.