Documentação

Introdução

A aplicação usa um compilador baseado numa gramática PEG.js 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
```

Definição de Pares Chave-Valor

Um par chave-valor é composto por dois elementos separadas por dois pontos (**:**).

A chave não pode conter espaços brancos (exceto entre o último caractere da *String* e o separador) nem qualquer outro caractere que não pertença ao alfabeto ou que não seja um *underscore*.

Por exemplo, `lorem_ipsum` é uma chave válida enquanto `lorem ipsum`, `lorem:ipsum` ou `lorem-ipsum` não são.

A única exceção é a diretiva `repeat`, que está entre plicas e que recebe como argumento um inteiro. Esta é responsável por gerar um *array* de objetos cujo comprimento é o dado por argumento.

Os valores podem ser um dos seguintes:

- Número
- *String*
- *Array*
- Booleano
- null
- Objeto DSL
- Função "Moustache"

Interpolações (Funções "Moustache")

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 execução, 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)}}'
```

De seguida estão explícitas todas as que se encontram atualmente disponíveis.

MOUSTACHE	TIPO	ARGUMENTO(S)	DESCRIÇÃO
objectID	`String`	`Nenhum`	Gera um ID aleatório com 24 bytes. Exemplo: "6048e87b9281fc9a1afe8e61"
guid	`String`	`Nenhum`	Gera um UUID aleatório. Exemplo: "3d16d5d0-4b11-4de8-9e26-6668b52d9219"
index	`Integer`	`Nenhum`	Retorna o índice atual do objeto gerado pelo 'repeat'.
boolean	`Boolean`	`Nenhum`	Gera um booleano aleatório.
firstName	`String`	`Nenhum`	Gera um nome próprio aleatório.
surname	`String`	`Nenhum`	Gera um apelido aleatório.
fullName	`String`	`Nenhum`	Gera um nome completo aleatório.
integer	`Integer`	`Min:: Integer,` `Max:: Integer`	Gera um inteiro aleatório entre Min e Max. Exemplo: integer(2,4) = 3

formattedInteger	`String`	`Min:: Integer,`
		`Max:: Integer,`
		`Pad:: Integer,`
		`Unid:: String`
			Gera um inteiro aleatório entre Min e Max, garante que tem pelo menos tantos algarismos quantos especificados no Pad e acrescenta uma *String* Unid no final.
			Caso não queira padding, pode colocar um 0 no 3º argumento e caso não queira colocar a unidade, pode colocar um "" no último argumento.
			Exemplo1: formattedInteger(2, 400, 3, "$") = "100$"
			Exemplo2: formattedInteger(1, 3, 3, "") = "002"

float	`Float`	`Min:: Integer,` `Max:: Integer`	Gera um número decimal aleatório entre Min e Max. O número de casas decimais do resultado vai ser igual ao do argumento com mais casas decimais. Exemplo: float(-180.4, 180.29) = -19.10

float	`Float`	`Min:: Float,`
		`Max:: Float,`
		`C:: Integer`
			Gera um número decimal aleatório entre Min e Max com um total de C casas decimais.
			Caso o número gerado acabe com 0s à direita na parte decimal, estes são omitidos.
			Exemplo: float(-180, 180, 2) = -19.11

formattedFloat	`String`	`Min:: Integer,`
		`Max:: Integer,`
		`C:: Integer,`
		`Pad:: Integer,`
		`Form:: String`
			Gera um número decimal aleatório entre Min e Max com um total de C casas decimais e garante que tem pelo menos tantos algarismos na parte inteira quantos especificados no Pad.
			Por fim, recebe um formato no argumento Form como "0#0#00?", onde o primeiro # é um caractere para separar cada 3 algarismos de inteiros, o segundo separa a parte inteira da decimal e ? é uma string a concatenar no fim (unidades).
			Caso não queira padding, pode colocar um 0 no 4º argumento e caso não queira colocar a unidade, pode acabar a *String* do formato logo a seguir aos últimos 00.
			Exemplo1: formattedFloat(2, 400, 3, 3, "0.0.00$") = "181.306$"
			Exemplo2: formattedFloat(2, 5, 2, 0, "0.0.00") = "2.05"

date	`String`	`Init:: String`	Gera uma data aleatória entre a data atual e a data argumento. A *String* do argumento tem de ter o formato "DD[./-]MM[./-]YYYY". O resultado é dado em versão JS raw. Exemplo: date("12-12-2100") = "2056-04-11T01:22:38.174Z"
date	`String`	`Init:: String,` `Form:: String`	Gera uma data aleatória entre a data atual e a data argumento, no formato dado. A *String* do 1º argumento tem de ter o formato "DD[./-]MM[./-]YYYY". Os formatos possíveis para o argumento Form são os seguintes: DD.MM.YYYY, DD.MM.AAAA, MM.DD.YYYY,

			MM.DD.AAAA, YYYY.MM.DD e AAAA.MM.DD, onde o . pode ser / - ou . Exemplo: date("12-12-2100","DD.MM.AAAA") = "09.12.2083"
date	`String`	`Init:: String,` `Fim:: String`	Gera uma data aleatória entre as datas argumentos. A *String* dos argumentos tem de ter o formato "DD[./-]MM[./-]YYYY". O resultado é dado em versão JS raw. Exemplo: date("12-12-2100","20-12-2100") = "2100-12-16T12:11:08.049Z"

date	`String`	`Init:: String,`
		`Fim:: String,`
		`Form:: String`
			Gera uma data aleatória entre as datas argumentos, no formato dado.
			A *String* dos 1ºs argumentos tem de ter o formato "DD[./-]MM[./-]YYYY".
			Os formatos possíveis para o argumento Form são os seguintes: DD.MM.YYYY, DD.MM.AAAA, MM.DD.YYYY, MM.DD.AAAA, YYYY.MM.DD e AAAA.MM.DD, onde o . pode ser / - ou .
			Exemplo: date("12-12-2100","20-12-2100","DD.MM.AAAA") = "18.12.2100"

lorem	`String`	`Num:: Integer,` `Parte:: String`	Gera Num palavras, frases ou parágrafos de *lorem ipsum*. A varíavel Parte tem de corresponder a "words", "sentences" ou "paragraphs". Exemplo: lorem(3,"words") = "mollit fugiat officia"

random	`Object`	`arg1:: Object,`
		`...,`
		`argN:: Object`
			Retorna aleatoriamente um dos argumentos passados à função.
			Exemplo: random("blue", true, false, 23, 17.56) = 23

position	`String`	`Nenhum`	Gera um conjunto de coordenadas cartesianas aleatórias. Exemplo: position() = "(67.95632, -55.44137)"
position	`String`	`[MinLat,MaxLat]:: [Float],` `[MinLon,MaxLon]:: [Float]`	Gera um conjunto de coordenadas cartesianas aleatórias, com limites superiores e inferiores. Exemplo: position([0.03,3],[-5,-2.4]) = "(0.26275, -4.03904)"
pt_phone_number	`String`	`Nenhum`	Gera um número de telemóvel português. Exemplo: pt_phone_number() = "911 154 239"
pt_phone_number	`String`	`Bool:: Boolean`	Gera um número de telemóvel português e com Bool==true é colocada a extensão. Exemplo: pt_phone_number(true) = "+351 911 154 239"
pt_district	`String`	`Nenhum`	Gera um distrito português aleatório. Exemplo: pt_district() = "Braga"
pt_district	`String`	`Def:: String,` `Coun:: String`	Gera o distrito português do concelho dado em Coun. O primeiro argumento tem de corresponder à *String* "county". Exemplo: pt_district("county","Braga") = "Braga"
pt_district	`String`	`Def:: String,` `Par:: String`	Gera o distrito português da freguesia dada em Par. O primeiro argumento tem de corresponder à *String* "parish". Exemplo: pt_district("parish","Tadim") = "Braga"
pt_district	`String`	`Def:: String,` `Cit:: String`	Gera o distrito português da cidade dada em Cit. O primeiro argumento tem de corresponder à *String* "city". Exemplo: pt_district("city","Barcelos") = "Braga"
pt_county	`String`	`Nenhum`	Gera um concelho português aleatório. Exemplo: pt_county() = "Braga"
pt_county	`String`	`Def:: String,` `Coun:: String`	Gera um concelho português aleatório do distrito dado em Dist. O primeiro argumento tem de corresponder à *String* "district". Exemplo: pt_county("district","Braga") = "Barcelos"
pt_county	`String`	`Def:: String,` `Dist:: String`	Gera o concelho português da freguesia dada em Coun. O primeiro argumento tem de corresponder à *String* "parish". Exemplo: pt_county("parish","Tadim") = "Braga"
pt_parish	`String`	`Nenhum`	Gera uma freguesia portuguesa aleatória. Exemplo: pt_parish() = "Tadim"

pt_parish	String	`Def:: String,` `Dist:: String`	Gera uma freguesia portuguesa aleatória do distrito dado em Dist. O primeiro argumento tem de corresponder à *String* "district". Exemplo: pt_parish("district","Braga") = "Tadim"
pt_parish	String	`Def:: String,` `Count:: String`	Gera uma freguesia portuguesa aleatória do concelho dado em Count. O primeiro argumento tem de corresponder à *String* "county". Exemplo: pt_parish("county","Braga") = "Tadim"
pt_city	String	`Nenhum`	Gera uma cidade portuguesa aleatória. Exemplo: pt_city() = "Braga"
pt_city	String	`Def:: String,` `Dist:: String`	Gera uma cidade portuguesa aleatória do distrito dado em Dist. O primeiro argumento tem de corresponder à *String* "district". Exemplo: pt_city("district","Braga") = "Barcelos"
political_party	Object	`Nenhum`	Gera um partido político aleatório e devolve um objeto com a abreviação e o nome correspondentes. Exemplo: political_party() = { "party_abbr": "Fr", "party_name": "Aliança de Paz" }
political_party	Object	`Cty:: String`	Gera um partido político aleatório do país dado em Cty e devolve um objeto com a abreviação e o nome correspondentes. Exemplo: political_party("Portugal") = { "party_abbr": "BE", "party_name": "Bloco de Esquerda" }
political_party_name	String	`Nenhum`	Gera o nome de um partido político aleatório. Exemplo: political_party_name() = "Aliança de Paz"
political_party_name	Object	`Cty:: String`	Gera o nome de um partido político aleatório do país dado em Cty. Exemplo: political_party_name("Portugal") = "Bloco de Esquerda"
political_party_abbr	String	`Nenhum`	Gera a abreviatura de um partido político aleatório. Exemplo: political_party_abbr() = "Fr"
political_party_abbr	Object	`Cty:: String`	Gera a abreviatura de um partido político aleatório do país dado em Cty. Exemplo: political_party_abbr("Portugal") = "BE"
pt_entity	Object	`Nenhum`	Gera uma entidade portuguesa aleatória e devolve um objeto com a sigla e a designação correspondentes. Exemplo: pt_entity() = { "abbr": "CMBRG", "name": "Câmara Municipal de Braga" }
pt_entity_abbr	String	`Nenhum`	Gera a sigla de uma entidade portuguesa aleatória. Exemplo: pt_entity_abbr() = "CMBRG"
pt_entity_name	String	`Nenhum`	Gera a designação de uma entidade portuguesa aleatória. Exemplo: pt_entity_name() = "Câmara Municipal de Braga"
soccer_club	String	`Nenhum`	Gera um clube de futebol aleatório. Exemplo: soccer_club() = "Liverpool FC"
soccer_club	Object	`Cty:: String`	Gera um clube de futebol aleatório do país dado em Cty. O nome do país pode ser dado em português ou em inglês e, para já, só são reconhecidos Portugal, Espanha, Itália, Inglaterra e Alemanha. Exemplo: soccer_club("Portugal") = "SC Braga"
actor	String	`Nenhum`	Gera um ator aleatório.
animal	String	`Nenhum`	Gera um animal aleatório.
brand	String	`Nenhum`	Gera uma marca aleatória.
buzzword	String	`Nenhum`	Gera uma buzzword aleatória.

capital	`String`	`Nenhum`	Gera uma capital aleatória.
car_brand	`String`	`Nenhum`	Gera uma marca de carro aleatória.
continent	`String`	`Nenhum`	Gera um continente aleatório.
country	`String`	`Nenhum`	Gera um país aleatório.
cultural_center	`String`	`Nenhum`	Gera um centro cultural aleatório.
gov_entity	`String`	`Nenhum`	Gera uma entidade governamental aleatória.
hacker	`String`	`Nenhum`	Gera um hacker aleatório.
job	`String`	`Nenhum`	Gera um trabalho aleatório.
month	`String`	`Nenhum`	Gera um mês aleatório.
musician	`String`	`Nenhum`	Gera um músico aleatório.
nationality	`String`	`Nenhum`	Gera uma nacionalidade aleatória.
pt_businessman	`String`	`Nenhum`	Gera um empresário português aleatório.
pt_politician	`String`	`Nenhum`	Gera um político português aleatório.
pt_public_figure	`String`	`Nenhum`	Gera uma figura pública portuguesa aleatória.
pt_top100_celeb...	`String`	`Nenhum`	Gera uma celebridade portuguesa aleatória.
religion	`String`	`Nenhum`	Gera uma religião aleatória.
soccer_player	`String`	`Nenhum`	Gera um jogador de futebol aleatório.
sport	`String`	`Nenhum`	Gera um desporto aleatório.
top100_celebrity	`String`	`Nenhum`	Gera uma celebridade aleatória.
weekday	`String`	`Nenhum`	Gera um dia da semana aleatório.
writer	`String`	`Nenhum`	Gera um escritor aleatório.

Primitiva 'repeat'

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 uma, desde um tipo JSON primitivo até um objeto complexo), bem como o número de cópias, ou intervalo de números.

```
1 nomes: [ 'repeat(150,200)': {
2     first: '{{firstName()}}',
3     last: '{{surname()}}'
4 }]
```

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 uma 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.

```
1 {
2     nomes: [ 'repeat(10)': '{{fullName()}}' ],
3     animais: [ 'repeat(20)': '{{animal()}}' ]
4 }
```

Primitiva 'unique'

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 querer que isso não 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`. 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:

```
// Primeiro caso
[ 'repeat(6)': {
    continente: unique('{{continent()}}'),
    país: unique('Country: {{country()}}'),
    aleatório: unique('{{random(1,2,3,4,5,6)}}')
} ]
// Segundo caso
[ 'repeat(10)': {
    continente: unique('{{continent()}}'),
    inteiro: unique('{{integer(5,20)}}'),
    aleatório: unique('{{firstName()}} {{surname()}}')
} ]

```

Primitivas map/filter/reduce

A gramática também disponibiliza uma implementação das ferramentas de programação funcional fundamentais - `map`, `filter` e `reduce`. O utilizador pode utilizar 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})
```

Execução de Código / Funções

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).

```
nome: "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 tem a 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.

Geração Difusa

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
}
```

Datasets

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 bilingue - 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 alternativamente
<!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;
- *buzzwords*
- partidos políticos;
- profissões
- políticos portugueses;
- religiões;
- top 100 celebridades;
- top 100 celebridades portuguesas.

Rotas Aplicacionais

As seguintes rotas REST são disponibilizadas como alternativa à interface gráfica:

- `POST` **/api/datagen/**
  O corpo tem de conter o modelo da gramática (DSL) em formato *raw*, isto é, não envolvido por aspas.

  Esta rota gera um objeto JSON com o dataset (também em JSON) e campos adicionais necessários para a sua inserção no Strapi.

- `POST` **/api/datagen/json**
  O corpo tem de conter o modelo da gramática (DSL) em formato raw, isto é, não envolvido por aspas.

  Esta rota retorna apenas o dataset resultante em formato JSON.

- `POST` **/api/datagen/xml**
  O corpo tem de conter o modelo da gramática (DSL) em formato raw, isto é, não envolvido por aspas.

  Esta rota retorna apenas o dataset resultante em formato XML.

Para além destas rotas, são também disponibilizadas rotas para obter os datasets utilizados na aplicação:

- `GET` **/api/dataset/:nome**
  Este `nome` terá de ser um dos seguintes:

  - actors
  - animals
  - brands
  - buzzwords
  - capitals
  - car_brands
  - continents
  - countries
  - cultural_centers
  - gov_entities
  - hackers
  - jobs
  - months
  - musicians
  - names
  - nationalities
  - political_parties

- pt_businessmen
- pt_districts
- pt_entities
- pt_politicians
- pt_public_figures
- pt_top100_celebrities
- religions
- soccer_clubs
- soccer_players
- sports
- top100_celebrities
- weekdays
- writers

Modelos Exemplo

O primeiro exemplo refere-se a um caso académico da representação, utilizando grafos, de 100 cidades portuguesas únicas, que atuam como os seus nodos. Estas têm:

- **id** que é dado por `c{N}` onde N é um número inteiro
- **nome**
- **população** que é um número inteiro entre 1500 e 550000
- **descrição**
- **distrito** da cidade

As arestas são as ligações entre as cidades (2000 no total):

- **id** que é dado por `l{id_ligação}-{id_origem}-{id_destino}`
- **origem** que é o **id** da cidade onde a ligação começa
- **destino** que é o **id** da cidade onde a ligação termina
- **distância** que é um número entre 5 e 600

```
<!LANGUAGE pt>
{
    cidades: [ 'repeat(100)': {
        id_cidade: 'c{{index(1)}}',
        nome: unique('{{pt_city()}}'),
        população: '{{integer(1500, 550000)}}',
        descrição: '{{lorem(1, "paragraphs")}}',
        distrito: '{{pt_district("city", this.nome)}}'
    }],
    ligações(gen) {
        var id = 1
        var cidades = this.cidades.map(x => x.id_cidade)
        var possiveis = cidades.flatMap((v, i) => cidades.slice(i+1).map( w => v + '|' + w ))
        var ligs = []

        for (var i = 0; i < 2000; i++) {
            let l = gen.random(...possiveis)
            possiveis.splice(possiveis.indexOf(l), 1)

            var split = l.split('|')
            ligs.push({
                id_ligação: `l${id++}-${split[0]}-${split[1]}`,
                origem: split[0],
                destino: split[1],
                distância: gen.float(5, 600)
            })
        }

        return ligs
    }
}
```

O segundo exemplo refere-se a um pequeno excerto de Autos de Eliminação que são, resumidamente, 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.

Neste exemplo serão abordadas apenas as seguintes secções desse documento:

- **tipo** da fonte de legitimação, que poderá ser "PGD/LC", "TS/LC", "PGD", "RADA" ou "RADA/CLAV"

- **fundos** que dependem do **tipo**:
  - se o tipo for "PGD/LC", "TS/LC" ou "PGD", só tem um fundo (entidade)
  - se não, tem entre 1 a 5 fundos (entidades)
- **classes** que são entre 2 e 5 e que dependem do **tipo**:
  - se o tipo for "PGD/LC" ou "TS/LC", tem um código (que pode ter 3 ou 4 níveis)
  - se não, pode ter um código, uma referência ou ambos

```
<!LANGUAGE pt>
{
  autoEliminação: {
        fonteLegitimação: {
            tipo: '{{random("PGD/LC", "TS/LC", "PGD", "RADA", "RADA/CLAV")}}'
        },
        fundos(gen) {
            if (["PGD/LC","TS/LC","PGD"].includes(this.fonteLegitimação.tipo))
                    return [gen.pt_entity()]
            else {
                    var arr = []
                    for (var i = 0; i < gen.integer(1,5); i++) arr.push(gen.pt_entity())
                    return arr
                }
        },
        classes: [ 'repeat(2,5)': {
            if (["PGD/LC","TS/LC"].includes(this.fonteLegitimação.tipo)) {
                código: 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) classe += '.' + nivel4
            return classe
                }
        }
            else {
                at_least(1) {
                    código(gen) {
                        var nivel1 = gen.random(...gen.range(100,950,50))
                            var nivel2 = gen.random(10,20,30,40,50)
                            var nivel3 = gen.integer(001,999)
                            var nivel4 = gen.random("01","02")

                            var classe = nivel1 + '.' + nivel2 + '.' + nivel3
                            if (Math.random() > 0.5) classe += '.' + nivel4
                            return classe
                        },
                    referência: '{{random(1,2,3,55,56)}}'
                }
          }
      }]
  }
}
```

Extras

O projeto está disponível no GitHub e é open-source, sujeito às licenças indicadas.

Para mais informações ou questões, é recomendado contactar os desenvolvedores do projeto, presentes na secção Sobre.