DataGen: JSON/XML Dataset Generator

Filipa Alves dos Santos
University of Minho, Portugal
a83631@alunos.uminho.pt

Hugo André Coelho Cardoso
University of Minho, Portugal
a85006@alunos.uminho.pt

João da Cunha e Costa
University of Minho, Portugal
a84775@alunos.uminho.pt

Válter Ferreira Picas Carvalho
University of Minho, Portugal
a84464@alunos.uminho.pt

José Carlos Ramalho
Department of Informatics, University of Minho, Portugal
jcr@di.uminho.pt

Abstract

In this document, it is described the steps towards DataGen's implementation.

DataGen is a versatile and powerful tool that allows for quick prototyping and testing of software applications, since currently too few solutions offer both the complexity and scalability necessary to generate adequate datasets in order to feed a data API or a more complex APP, enabling their testing with appropriate data volume and complexity.

DataGen's core is a Domain Specific Language (DSL) that was created to specify datasets. This language suffered several updates: repeating fields (with no limit), fuzzy fields (statistically generated), lists, high order functions over lists, custom made transformation functions. The final result is a diversified algebra that allows the generation of very complex datasets coping with very convoluted requirements. Throughout the paper, several examples of the possibilities will be given.

After generating a dataset, DataGen gives the user the possibility to generate a RESTFul data API with it, creating a running prototype.

This solution has already been used in real life cases, described with more detail throughout the paper, in which it was able to create the intended datasets successfully. These allowed the application's performance to be tested and for the right adjustments to be made.

The tool is currently being deployed for general use.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Theory of computation → Grammars and context-free languages; Information systems → Open source software

Keywords and phrases JSON, XML, Data Generation, Open Source, REST API, Strapi, JavaScript, Node.js, Vue.js, Scalability, Fault Tolerance, Dataset, DSL, PEG.js, MongoDB

Digital Object Identifier 10.4230/OASIcs.SLATE.2021.5

1 Introduction

Every application and software developed should be thoroughly tested before release, in order to determine the system's ability to withstand realistic amounts of data and traffic, and that implies the usage of representative datasets that fit its use cases. The creation of said datasets is a laborious and drawn out process, as it implies firstly generating the test data in some way, in a file format compatible with the system. As it stands, there are currently

10th Symposium on Languages, Applications and Technologies (SLATE 2021).
Editors: Ricardo Queirós, Mário Pinto, Alberto Simões, Filipe Portela, and Maria João Pereira; Article No. 5; pp. 5:1–5:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

no efficient, intuitive and scalable tools to do this and so, developers often end up having to create test records either by hand, which is incredibly inefficient and time-consuming, or by using existing tools with some clever tactics to manage their shortcomings. As a matter of fact, many projects are not able to progress to the development stage due to the lack of adequate and sufficient data [3].

Even with a reliable generation tool, the user might want to have control over the data in the resulting records, being able to observe and manipulate it freely, through CRUD requests, and adapting it however they want. Though there are products capable of doing this - for example the package json-server, which creates a full fake REST API -, its usage entails the user manually starting the application every time they want to edit the dataset's contents and ensuring the data's format is compliant with the software they're using, which ends up requiring a lot of extra effort and setup on the user's side.

As such, the team came up with the idea of coupling the generation process and the REST API, in a way that allows the user to automatically generate data compatible with an integrated RESTful API - for which the software Strapi was chosen, as will be explained in more detail in section 3.4 -, allowing the user to load the records into an API server and perform CRUD operations over it, either through the user interface or via HTTP requests.

This paper will cover the development of the resulting application, DataGen - a more versatile and powerful tool for data generation, according to the user's specifications, and subsequent handling -, seeking to explain the decisions that were taken for its architecture, as well as all the implemented functionalities.

2 Related Work

User privacy concerns [13] have been a major topic of discussion over the last decades, which lead to the creation of strict regulations regarding the way sensitive data should be handled, such as the EU's General Directive on Data Protection GDPR [2]. These regulations keep entities from disclosing private and sensitive information, which in turn hurts new growing ideas and projects that would require access to similar data. As it stands, not only is the lack of available data a big problem in this context, as are the data sharing agreements themselves, as their ratification tends to take, on average, a year and a half [10], which proves to be fatal for many small and upcoming companies.

To circumvent these concerns, organisations have been increasingly adopting synthetic data generation [16], an approach that was originally suggested by Rubin in 1993 [1] in order to create realistic data from existing models without compromising user privacy. The prospect of being able to create viable information that does not relate to actual individuals is a very enticing solution for the privacy conundrum. If perfected, it could potentially provide the capacity of generating sizeable amounts of data according to any use cases, bypassing the need to access real users' information. As such, this approach has been increasingly adopted and put to the test, in order to measure its efficiency with real-life cases [5, 14, 18].

Dataset generation has become a requisite in many development projets. However, the majority of developed tools produce datasets for specific contexts like intrusion detection systems in IoT [4], crops and weed detection [6], vehicular social networks based on floating car data [12], 5G channel and context metrics [17], GitHub projects [9], to cite a few. Many others exist in different contexts like medicine, bioinformatics, weather forecasts, color constancy, market analysis, etc.

Most of the researched tools are domain specific. The team's goal is to create a generic tool but powerful enough to generate datasets for several and different contexts and with many levels of complexity. There are some tools available, many online, but they cover the generation of simple datasets, many times flat datasets.

The main source of inspiration for this project was an already existing web application called "JSON Generator", developed by Vazha Omanashvili [11], as it is the most interesting dataset generation tool that was found. It features a DSL (Domain Specific Language) that's parsed and converted into a JSON dataset, allowing the user to generate an array of objects that follow some predefined structure, specified in the DSL.

In terms of utility, it is a very powerful tool that can generate very complex data structures for any application with relatively little effort. However, it had some shortcomings which initially inspired the team to develop an alternative solution that attempts to address them.

Specifically, these shortcomings are:

1. Limited size for the 'repeat' statement (100 total). Arguably, the size of the dataset itself is one of the most important features to take into account. For applications on a larger scale, having a small amount of array elements does not allow for more realistic tests, as very few, if any, that are developed in a production environment use as little as 100 entries created by the aforementioned directive.
2. It only generates JSON. Despite being one of the most popular file formats, there are many others that could be useful to the user as they might want the data to be in a different format for their application without converting the JSON themselves, such as XML (another open text format [19]). This allows for further flexibility and expands the possible use cases that it provides.
3. Does not generate a RESTful API for the dataset. Many users may optionally want their newly generated dataset hosted and exposed by a RESTful API for direct usage in their web applications, or to download a custom one created specifically for their dataset for later deployment on a platform of their choosing.
4. Does not take into account fuzzy generation. Some elements of a dataset may not be deterministic and are represented by probabilities. For example, there may a field that exists only if another property has a specific value and the application should be able to take that into account.
5. It does not have much data available. For instance, the user might want to use names from a list of famous people for their dataset, as it allows for a more realistic generation and consistency, which this tool currently does not provide.
6. It is not multilingual. The data that this application uses is only available in English, it would be more user-friendly to give them the choice to use their preferred language for the dataset instead of forcing it to just one.
7. Does not take into account integration on applications. The generation and download of a dataset requires the consumer to use the website's interface - this is not ideal as many applications may want to use HTTP requests to automate this process for internal usage.
8. Does not support functional programming features. Functions such as 'map', 'reduce' and 'filter' that are staple in the functional paradigm due to their simplicity and effectiveness are not present in this application. This would allow the user to chain statements and transform fields to the result they want, granting the application the ability to deal with more complex use cases.

With clear goals and an initial application to take inspiration from, the team proceeded to the development stage by deciding on its architecture (i.e. programming languages, frameworks, external tools, etc), which will be explained in the following sections.

3 DataGen Development

Building the application from the ground up requires a divide and conquer approach, since having a monolithic single server architecture will lead to a less stable user experience, due to the lack of scalability.

Having the application compartmentalized in multiple servers, each with their specific function, allows for a much more sensible and balanced architecture since it leaves room for the possibility of individually upgrading each of them, leading to a much higher scalability and fault tolerance, as the failure of one component does not compromise the functionality of the entire application, allowing for quicker and easier routine maintenance.

The following subsections will explain how the current architecture was achieved and the technological decisions behind each of the components.

3.1 Architecture

The picture below shows the general scope of the architecture, albeit simplified. This architecture allows for major upgrades, such as load balancers on both the back and front-end since they are both **stateless** (the JWT approach allows the servers to not maintain sessions with each user's state) and using MongoDB as a distributed database - sharded cluster.

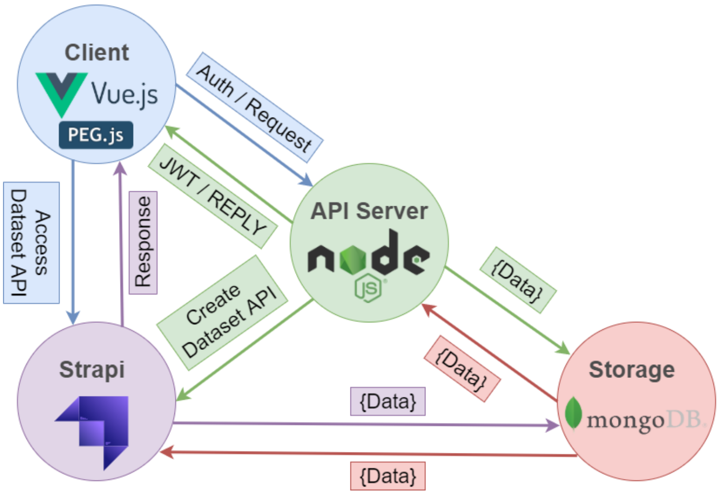

Figure 1 Current architecture of DataGen

3.2 Front-End

The first component needed for the application is the front-end server, which is responsible for building and showing the website's interface to the user, making it the entry point to the application and its scripted behaviour.

3.2.1 Grammar

The application uses a PEG.js grammar-based parser [8] [15] to process the user's input and generate the intended dataset. The aforementioned grammar defines a domain-specific language (DSL), with JSON-like syntax, providing many features that allow for the generation of complex and diverse datasets. These features include relational and logic capabilities, providing means for the datasets to satisfy several forms of constraints - which push towards the use of some declarative framework for this specification -, as well as functional capabilities, allowing for easier management and processing of certain properties of the datasets.

The first and most fundamental of said features is the JSON-similar syntax - the user can specify key-value properties, where the value may take any of the basic JSON types and data structures, from integers to objects and arrays. The user may also nest these to create a structure with any depth they may want.

```
name: {
  first: ["John", "William"],
  last: "Doe"
},
age: 21
```

To specify the size of the dataset, or a nested array, there is the **repeat** statement, where the user indicates the structure that they want replicated (which may range from a primitive JSON type to a complex object), as well as the number of copies, or range of numbers.

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

They may also specify as many collections as they want in a single model (collections are the key-value properties on the uppermost level of the model) and the application will return the resulting dataset in json-server syntax - an object with a property for each collection. During the parsing of the input, the application recursively builds both the final dataset and the Strapi model for the specified structure, concurrently, in order to allow for posterior integration in a RESTful API.

```
{
  names: [ ’repeat(10)’: ’{{fullName()}}’ ],
  animals: [ ’repeat(20)’: ’{{animal()}}’ ]
}
```

To define the value of a property, the user may also use interpolation. To access an interpolation function, it must be encased in double curly braces. There are two types of interpolation functions:

- functions that generate spontaneous values during execution, according to the user’s instructions - for example, there is a random integer generation function where the user must provide a range of values for the intended result:

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

- functions that return random values from a group of datasets incorporated in the application behind an API, where each dataset has information of a given category, for example names and political parties.

```
name: '{{fullName()}}',
party: '{{political_party()}}'
```

These interpolation functions may also be interwoven together and with normal strings to generate more structured strings, such as addresses. Some of them take arguments, in which case the user can either manually introduce the value or reference another property defined above, through a variable **this**, allowing them to establish relations between properties.

```
parish: '{{pt_parish()}}',
district: '{{pt_district("parish", this.parish)}}',
address: 'St. {{fullName()}}, {{pt_city("district", this.district)}}'
```

In regard to the API of datasets, the team did an extensive search for useful datasets, used the well-structured ones it found and salvaged whatever information it could from others that were maybe less organized, processing this information to remove errors and normalize its content, before joining it with other data of the same topic to create bigger, more complete datasets for the user to use.

The team also created some original datasets by hand, for topics deemed appropriate, and manually introduced bilingual support, for both portuguese and english, in all datasets made available in the application, in order to let the user choose whichever language suits best their goal. To indicate their language of choice, the user's model must start with the following syntax:

```
<!LANGUAGE pt> or <!LANGUAGE en>
```

Currently, DataGen has support datasets of all the following categories: actors, animals, brands, buzzwords, capitals, cities, car brands, continents, countries, cultural landmarks, governmental entities, hackers, job titles, months, musicians, names, nationalities, political parties, portuguese businessmen, portuguese districts, cities, counties and parishes, portuguese public entities, portuguese politicians, portuguese public figures, portuguese top 100 celebrities, religions, soccer clubs, soccer players, sports, top 100 celebrities, weekdays, writers.

The grammar also makes available a feature named **unique()**, to which the user may provide an interpolation function, or a string interpolated with such a function, as an argument. **unique** guarantees that the interpolation functions on which it is applied always return unique values. This is especially relevant when it comes to interpolation functions that fetch random data from the support datasets inside a **repeat** statement, as there is no guarantee that there won't be duplicates among the fetched records and the user might not want that.

As such, **unique** only has any effect when applied on dataset interpolation functions or with **random** (which can be observed in one of the examples from last page). As long as it's one of those (possibly interpolated with normal strings) and there are sufficient distinct entries for the entire **repeat** statement, this tool guarantees that all objects in the resulting dataset will have a different value in the property in question. If the user uses a string with more than one interpolation function, there is also no effect - there may be repeated combinations in the end.

Below are two examples: the first one depicts the correct usage of the **unique** feature; the second shows instances of a wrong approach (not enough distinct values for the repeat statement; not a dataset interpolation function or the random function; more than one interpolation function) that will either not work or not assure any mutually exclusive guarantee for the resulting values:

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

[ ’repeat(10)’: {
  continent: unique(’{{continent()}}’),
  int: unique(’{{integer(5,20)}}’),
  name: unique(’{{firstName()}} {{surname()}}’)
} ]
```

Back to the properties of the model, the user may also use JavaScript functions to define their value. There are two types of functions: signed functions, where the name of the method corresponds to the key of the property, while the result of the body of the function translates to its value, and anonymous arrow functions, which are used to indicate solely the value of a property (the key needs to be precised separately beforehand).

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

Inside these functions, the user is free to write JavaScript code that will be later executed to determine the value of the property. This way, more complex algorithms may be incorporated into the construction logic of the dataset, allowing for a more nuanced and versatile generation tool. Given that the user has access to the whole Javascript syntax, they may make use of relational and logical operators to elaborate conditions on the intended data, as well as functional methods (for example "map" or "filter", which Javascript implements).

Inside these blocks of code, the user has full access to any property declared above in the DSL model, through the variable **this**, as well as any interpolation function available in the parser, through a **gen** variable - whenever using a function, the user must declare this argument in its signature, which they may then use inside to run said interpolation functions. All of this can be observed in the example above.

The grammar also allows fuzzy generation of properties, i.e. constraining the existence of certain properties based on logical conditions or probabilities. As of now, the grammar has four different tools for this purpose:

- **missing/having** statements - as an argument, they receive the probability of the properties inside (not) existing in the final dataset; this probability is calculated for each element, originating a dataset where some elements have said properties and others don’t:

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

- **if... else if... else** statements - these work just as in any programming language: the user can use relational operators and other conditional statements to create conditions and string multiple of them together with the help of logical operators. The final object will have the properties specified in the first condition that evaluates to true (or eventually none of them, if all conditions prove to be false). In these conditions, similar to the functions, the user has unrestricted access to all properties declared above in the DSL model, as well as the interpolation functions, which gives them the ability to causally relate different properties:

```
type: ’{{random("A","B","C")}}’,
if (this.type == "A") { A: "type is A" }
else if (this.type == "B") { B: "type is B" }
else { C: "type is C" }
```

- the **or** statement - the grammar makes available this logical operator for quick prototyping of mutually exclusive properties, where only one will be selected for each object (note that it doesn’t make sense to create an **and** statement, since that translates to simply listing the wanted properties normally in the DSL model):

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

- the **at_least** statement - inside this, the user writes a set of properties and gives the statement an integer argument specifying the minimum number of those properties that must be present in the final object. The parser selects that number of properties randomly from the given set:

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

Finally, the grammar also provides an implementation of the fundamental functional programming features - **map**, **filter** and **reduce**. The user may chain together one or several of these functions with an array value (from any of the various array-creating features made available). Shorthand syntax is not allowed, so curly braces must always be opened for the code block. Aside from that, these features work exactly like the native Javascript implementations: the user may either declare the function inside or use anonymous syntax for the variables; they may declare only the current value or any of the additional, albeit less commonly used, variables. Examples of several possible usages of these features can be observed below:

```
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] }),
combined: 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 Interoperability

After processing the model, the parser generates an intermediary data structure with the final dataset, which can then be translated to either JSON or XML, according to the user's preference. The parser also generates another data structure with the corresponding Strapi model, for possible later integration in its RESTful API.

Note that the application's purpose is to create datasets according to the user's instructions, in either JSON or XML. Although the model specification may involve Javascript code, under the form of functions or conditions, as explained in the previous subsection, this does not correlate to the target application whatsoever. DataGen merely generates test datasets - it can be used for any kind of application that accepts data in JSON/XML format, whether it be an application written in Javascript, Python, C++ or some other language.

3.2.3 Client-Side Generation

This project was developed with the intent of being a web application, more specifically a website with user-friendly features. A server-sided approach would imply parsing the DSL models on the back-end server, which wouldn't be sensible as the created PEG.js parser doesn't require access to any private services (i.e. databases) hidden by the back-end itself.

Therefore, a client-sided approach makes the most sense for this application in particular, freeing resources (mainly the CPU and memory modules) from the back-end server and shifting the computation of the generated dataset to the client in their browser, using the back-end as an API server.

There are many frameworks aimed at client-sided browser experiences, however, it was decided that Vue.js would be the most adequate for this application. It allows for reactive two-way data binding - connection between model data updates and the view (UI) which creates a more user-friendly experience, since it shows changes on the DOM as they occur, instead of forcing a reload on the page (as in the traditional served-sided approach). Other reasons such as flexibility - on the components' styling and scripted behaviour - and performance - it's more efficient than React and Angular - were also a deciding factor on picking this specific framework.

After deciding which framework to use, the team started developing the interface itself, which currently has the following features:

- Authentication - it uses JWT (JSON Web Tokens) that are sent in the 'Authorization' header on every HTTP request that needs to be validated by the back-end (i.e accesses restricted user data);
- Generation and download of the dataset and/or its API - as previously mentioned, it uses a PEG.js parser for converting the DSL into JSON or XML, as well as Strapi for the API (which will be explained in section 3.4);
- Saving DSL models - authenticated users may save their created models and optionally make them available for others to use, being able to do CRUD operations on those they own;
- Documentation - since the DSL has a defined structure, it is essential that the user has access to these guidelines at any time;
- Team description - the user may want to contact the team directly so there is a dedicated page for them to find all of this information and a brief description of the team.

The front-end needs to access persistent information such as user data and saved models which is accessible through the back-end's RESTful API, viewed in more detail in section 3.3.

3.3 Back-End

The application needs a back-end server for multiple purposes, namely storing data, authenticating users and generating the API for the datasets.

None of the above require intensive computational power for each request sent by the user, which was one of the main reasons why the team chose a Node.js server - it is built to deal with any incoming request that is not CPU intensive due to its single-threaded, event-driven, non-blocking IO model - and because it is scalable, has good performance in terms of speed and has a wide selection of packages (available on the **npm** registry).

For storing all the data that needs to be persistent, the back-end server accesses a MongoDB server instance, which was chosen due to its scalability (the data is not coupled relationally, which means that each document may be in a different node instance without any conflicts since they are self-contained) and direct compatibility with Node.js since they both accept JSON documents.

Currently the application uses three collections on the MongoDB database:

- **users** - stores all user specific data, which is their name, email, password (hashed) and the dates of register and most recent access;
- **models** - stores all DSL models and whom (user) they belong to, their visibility (public or private), title, description and register date;
- **blacklist** - stores users' JWT tokens after they log-out and their expiry date so that they are automatically removed from the database after they expire.

Authenticating the user allows them to access their saved DSL models and perform CRUD operations on them. Due to its Node.js integration, a JWT (JSON Web Token) approach was the chosen strategy to authenticate a user - after they submit their credentials, the system compares them to the ones saved on the database and if they match, the token is returned in the HTTP response. This token is needed for any further request that accesses critical information for that same user and expires after a short time for precaution and safety. After they log-out, it is added to a blacklist to provide extra security, since it does not allow for a user that got access to another user's JWT (if they log-out before the short expiration date) to submit requests signed with it.

Generating the API is a more complex process and has a dedicated subsection (3.4) which explains the steps followed in order to obtain a fully functional REST API for any generated dataset.

3.4 Strapi API

DataGen also provides another important functionality, which is generating a data API from the dataset previously created. It's a useful feature since a lot of users may want to perform CRUD operations on the data they requested or even utilize the API themselves for further work.

The tool chosen to create this API was Strapi [20], one of the best content management systems currently. Strapi automatically generates a REST API and allows multiple APIs to run simultaneously. It's also very simple to configure and supports different database systems like PostgreSQL, MySQL, SQLite and MongoDB, being that the latter was the one used in this project. JSON-server was also considered as a tool but lacked scalability, as it only allows a single API to run at a time, which wouldn't be ideal at all. However, Strapi presented its own set of challenges, like the difficult way in which it stores structured data (an array, for example) and how data is imported, all of which were surpassed successfully.

The process of building the API begins within the grammar section of the project, since the Strapi model is written in a recursive way, at the same time the dataset itself is being built. This strategy was a big time save in terms of the program's execution. For example, any time an array is encountered, because Strapi doesn't have its own type to represent it, a component is created with the array's elements and a reference to that component is written in the model itself. This recursive process keeps on going with this same logic until it reaches the root, which corresponds to the collection.

After the model is created, this data is redirected to an auxiliary application that processes and rearranges it to be in the usual Strapi format. The data consists in the finished model and also an array filled with all the components created. At this point, the user can download a zipped version of the API model, if they so intend, and easily run it on their personal device.

Furthermore, DataGen populates the newly created API with the generated dataset through a number of POST operations. Because of Strapi's lack of methods for importing whole files as data, this cycle of POST requests was the solution found to provide a temporary populated API REST, with all the standard HTTP methods functional.

4 Results

One of the priorities during development was to test DataGen with real cases from early on, in order to not only validate the concept and its operability, but also to observe what kind of restrictions and requests were more frequent in the creation of test datasets, as a means to gather reliable external feedback on useful capabilities that should be implemented.

The team made contact with other parties and received requests to create test datasets for real systems, using DataGen, which involved the usage of complicated generation logic with many restrictions. These opportunities helped further DataGen's growth, as new ideas arised from the analysis of the requirements and were brought to life in the application, as well as proved the application's credibility, given that the results obtained were adequate and very positive.

In the interest of keeping this paper concise, it will be shown only the most important parts of one of the most complex of these application cases, that of elimination records [7].

Elimination records are a structure that must be created and carefully filled out in order to safely eliminate documentation that reaches the end of its administrative conservation deadline. This is an increasingly important tool nowadays, since most public information has shifted to being stored in digital format and the correct method for storing such data is often not followed, which increases the risk of long-term information loss. In order to correct this, the deletion of outdated documentation is just as important as the storage of new one.

The generation of these documents implies a complex logic, with many properties that directly relate between themselves according to their values and others whose value must belong to a very rigid group of possibilities. Each record has a legitimation source, whose type can take one of five different string values. According to the record’s source type, its funds (public entities) vary from a single entity, in some cases, to an array of several:

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

Moving on, each record has an array of classes. In case the legitimation source’s type is "PGD/LC" or "TS/LC", each class has a code; else, it has either a code, a reference or both. The class code itself can be composed by 3 or 4 levels, given that each level follows its own categorization:

```
classes: [ ’repeat(2,5)’: {
  if (["PGD/LC","TS/LC"].includes(this.legitimationSource.type)) {
    code(gen) {
      var level1 = gen.random(...gen.range(100,950,50))
      var level2 = gen.random(10,20,30,40,50)
      var level3 = gen.integer(1,999,3)
      var level4 = gen.random("01","02")

      var class = level1 + ’.’ + level2 + ’.’ + level3
      if (Math.random() > 0.5) class += ’.’ + level4
      return class
    }
        }
  else {
    at_least(1) {
      code(gen) { (...) //equal to the function above },
      reference: ’{{random(1,2,3,55,56)}}’
    }
  }
} ]
```

There are also year properties that must belong to the last 100 years:

```
yearStart: ’{{integer(1921,2021)}}’,
yearEnd(gen) {
  var year = gen.integer(1921,2021)
  while (year < this.yearStart) year = gen.integer(1921,2021)
  return year
}
```

Finally, there are two related fields, number of aggregations and the corresponding list, where the size of the list must correspond to the number indicated:

```
numberAggregations: ’{{integer(1,50)}}’,
aggregations: [ ’repeat(this.numberAggregations)’: {
  code: ’{{pt_entity_abbr()}} - {{integer(1,200)}}’,
  title: ’{{lorem(3,"words")}}’,
  year: ’{{integer(1921,2021)}}’,
  if (["PGD/LC","TS/LC"].includes(this.legitimationSource.type)) {
      interventionNature: ’{{random("PARTICIPANT","OWNER")}}’
  }
} ]
```

5 Conclusion

Along the paper it was discussed the development of a multilingual data generator, with built-in REST API integration. The intent behind this project was to create a versatile and powerful tool that would allow for quick prototyping and testing of software applications, a very important and common subject that seems to go surprisingly unnoticed, despite its vast relevance.

Be it either small-scale projects of university students or big, complex company software, every application should be thoroughly tested along its development, which requires the leading team to create realistic data in order to populate the system. Even today, this process is very poorly optimized, which often leads either to very time-consuming manual generation or, even worse, to a scarce and inneficient testing of the system, with few records, possibly leading to wrongful conclusions, unnoticed bugs and dangerous bottlenecks.

As such, DataGen emerges as a quick and easy to use application that allows the user to swiftly prototype a data model according to their use cases and put their system to practice with a newly-generated dataset, with accurate and realistic values, automating the generation process and facilitating the user’s role in it, ultimately enhancing the user’s experience and allowing more time and resources to go towards the project itself.

DataGen was thoroughly experimented with real-life cases and proved to be capable of creating complex and sizeable datasets for third party applications. The product will very soon be put in a production environment and made available for the general public, after a laborious and successful development phase.

6 Future work

This platform will be open-source and its contents will be uploaded to GitHub. The next step for the application itself is putting it in a production environment, to be made available for anyone that may want to use it.

As for the grammar, the team intends to develop an user-friendly personalized grammar checker that analyzes the user's DSL model and, in the presence of errors, communicates what they are, exactly where they occur and how to fix them, in a simple and clear way, in order to enhance the user's experience and make the application easier to use.

Extensive documentation on all the functionalities provided is also under development, along with examples on how to use them, in order to guide the user since the application uses a DSL. Without it, the user may be overwhelmed by the amount of features they must learn by themselves. This documentation will be made available in the website and may eventually be downloaded in PDF format, if the user so wishes.

References

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.