Test Data Generator

DOCUMENTATION

Binod Shrestha, Dipesh Bhatta, Shashwat Mehta, Yashasvi Pant

DEERWALK SERVICES PVT. LTD. | SIFAL, KATHMANDU

Domain Structure

Domain structure is defined in JSON file. For a particular domain, we have particular JSON file. The JSON file is named by the name of the domain. The following are the keywords used in JSON:

1. output_delimiter: To specify the delimiter of the output csv file. Any single character delimiter is supported. If not specified, the default - ',' is used.

2. tables: Here tables are stored as JSON Array. It has following keywords in each element of JSON array:

2.1 name: To specify the table name. Alphanumeric with dashes and underscores are acceptable, spaces are not.

2.2 source: To specify table source (if available). It has following elements as keyword:

2.2.1 path: It is the path of the table source file.

2.2.2 delimiter: It is the delimiter of the input table source file. Any single character delimiter is supported. If not specified, the default - ',' is used.

2.3 dependency: To specify the table name (if any) that the current table depends on.

2.4 fields: Here, fields of the table are stored as JSON Array. It has the following keywords in each element of JSON Array:

2.4.1 field_name: To specify the field name. Alphanumeric with dashes and underscores are acceptable, spaces are not.

2.4.2 type: To specify type of field. It supports **integer, String, date and double.**

2.4.3 source: To specify the input source of data of a particular field. It can have following elements:

2.4.3.1 array: If the input source of data is stored as array.

2.4.3.2 relation: If the input source of data is extracted from field of another table (table relation case).

2.4.3.3 path: If the input source of data is extracted from file of specified path.

2.4.3.4 index: If the table source is specified give it's corresponding column index as **0, 1, 2..** etc.

2.4.4 unique: To generate unique integer. It has value **true or false**.

2.4.5 autoincrement: To generate auto-incrementing integer. It has value **true or false**.

2.4.6 lower_range: To state lower range of integer, date and double.

2.4.7 upper_range: To state upper range of integer, date and double.

2.4.8 date_format: To specify the format of date data type.

The following is the example of sample domain:

```
{
  "output_delimiter": ",",
  "tables": [
    {
      "name": "table_name1",
      "dependency": "table_name2",
      "fields": [
        {
          "field_name": "name",
          "type": "String",
          "source": {
            "array": "{amit, dipesh, dinesh, dipak, anita, deepika,
                      dipti, dinank}"
          },
          "unique": "false"
        },
        {
          "field_name": "Age",
          "type": "integer",
          "source": {
            "relation": "table_name2.Age"
          }
        },
        {
          "field_name": "id",
          "type": "integer",
          "autoincrement": "true",
          "lower_range": "4",
          "upper_range": "125"
        },
        {
          "field_name": "caste",
          "type": "String",
          "source": {
            "relation": "table_name2.caste1"
          }
        },
        {
          "field_name": "Address",
          "type": "String",
          "source": {
            "path": "src/main/resources/address.csv"
          }
        },
        {
          "field_name": "DOB",
          "type": "date",
          "source": {
```

```
          "relation": "table_name2.DOB"
        }
      },
      {
        "field_name": "salary",
        "type": "double",
        "source": {
          "relation": "table_name2.salary"
        }
      }
    ]
  },
  {
    "name": "table_name2",
    "source": {
      "path": "src/main/resources/input/table_nam2SourceFile.csv",
      "delimiter": ";"
    },
    "fields": [
      {
        "field_name": "name",
        "type": "String",
        "source": {
          "index": "0"
        }
      },
      {
        "field_name": "DOB",
        "type": "date",
        "upper_range": "2055-12-12",
        "lower_range": "1995-25-05",
        "date_format": "yyyy-dd-MM"
      },
      {
        "field_name": "Age",
        "type": "integer",
        "lower_range": "50",
        "upper_range": "125"
      },
      {
        "field_name": "id",
        "type": "integer",
        "autoincrement": "true",
        "lower_range": "1",
        "upper_range": "125",
        "unique": "true"
      },
      {
        "field_name": "caste1",
        "type": "String",
        "source": {
          "index": "1"
        }
      },
      {
        "field_name": "salary",
        "type": "double",
```

```
            "lower_range": "5000.00",
            "upper_range": "200000.00"
          }
        ]
      },
      {
        "name": "table_name3",
        "source": {
          "path": "src/main/resources/input/table_nam3SourceFile.csv",
          "delimiter": ","
        },
        "dependency": "table_name2",
        "fields": [
          {
            "field_name": "id",
            "type": "integer",
            "autoincrement": "true",
            "lower_range": "1",
            "upper_range": "125",
            "unique": "true"
          },
          {
            "field_name": "name",
            "type": "String",
            "source": {
              "index": "0"
            }
          },
          {
            "field_name": "Address",
            "type": "String",
            "source": {
              "path": "src/main/resources/address.csv"
            }
          },
          {
            "field_name": "caste",
            "type": "String",
            "source": {
              "relation": "table_name2.caste1"
            }
          },
          {
            "field_name": "salary",
            "type": "double",
            "lower_range": "100.00",
            "upper_range": "200000.55"
          }
        ]
      }
    ]
}
```

Query Structure

We have used ANTLR to generate lexer and parser. The parser is then used to generate parse tree. The parse tree thus generated is then visited to validate the conditions and to generate the required data. Each query consists of sub-queries or conditions or both.

Currently our custom Query Language supports AND/and and OR/or as combination operator for multiple sub-queries and conditions. It also supports the nested query of any level. We can also provide different operators in conditions to explain our required data properly. The operators are:

```
=       : equals to
>       : greater than
<       : less than
>=      : greater than or equal
<=      : less than or equal
!=      : not equal to
( and ) : opening and closing braces for nested queries
```

The custom Query Language currently supports the integer, float and string datatypes.

We can also use pipe (|) operator to hit the query in multiple tables.

Examples:

For the give domain configuration shown in previous section, we could use the queries like:

```
1. table_name1.id>10
2. table_name1.id<120 and table_name1.name="d.*"
3. table_name1.name=".*a" and (table_name1.salary>1000 or
   table_name1.DOB<="2001/05/05")
4. and many more
```

To hit the query in multiple tables (eg. for `table_name2` and `table_name3`):

```
1. table_name2.name="a.*" | table_name3.id>20
2. table_name2.name="a.*" and table_name2.salary<10000 | table_name3.caste="c.*"
3. and many more
```

Data Blinding Rules

1. We have following tokens to blind data:

- **String manipulation token**: for manipulating data in string: **rotate and encrypt**.
- **Direction token**: for the rotating direction of string: **ccw, counterclockwise and anticlockwise**.
- **Increase token**: to increase integer, float and date: **increase, inc and add**.
- **Decrease token**: to decrease integer, float and date: **decrease, dec, subtract, sub, reduce**.
- **Date Units**: **DAY, MONTH, WEEK, and YEAR**

2. Syntax:

For strings:

```
"<fieldName>":"<manipulation token> by <value> <direction token>"
```

```
eg:    "name":"rotate by 3 counterclockwise"
```

For integers, floats and dates:

```
"<fieldName>":"<increase token/decrease token> by <value> <date Units>(in case of Date)"
```

```
eg:    "id":"decrease by 10"
```

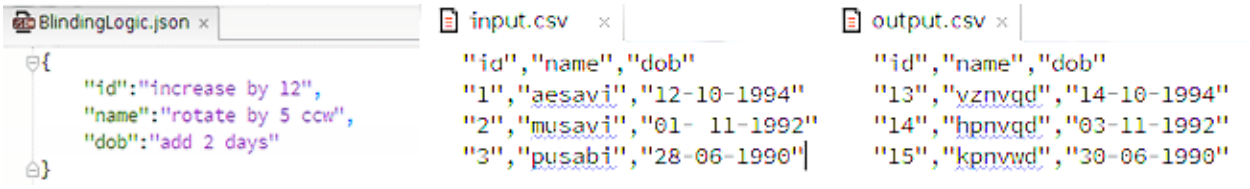
```
eg:    "dob":"add 7 days"
```

3. Every field of the data should be defined either with some value or with null value.

```
eg:    "name":"rotate by 0"
```

```
or     "name":"rotate by 5 ccw"
```

Below is an example of data blinding in action,



Front End

The front end is pretty self-explanatory and straightforward to use. Utmost focus has been put to create a satisfying user experience; and proper navigation, languages and help texts have been put.

As you probably already know, the front end is a web app, developed in `grails 2.4.4`, using `jdk 1.7`.

Make sure you are connected to the Internet, and run the following command to clone the complete project from following git repository,

```
git clone https://github.com/shashwatblack/testdatagenerator101.git
```

This will also download the project back end as jar, some grails dependencies and all web dependencies. These have been included in the project folder itself to avoid having to download them during every usage session.

Let's move into the directory,

```
cd testdatagenerator101/
```

Now with `$GRAILS_HOME` and `$JAVA_HOME` properly defined in your system, start up the grails server and run the app using the following command,

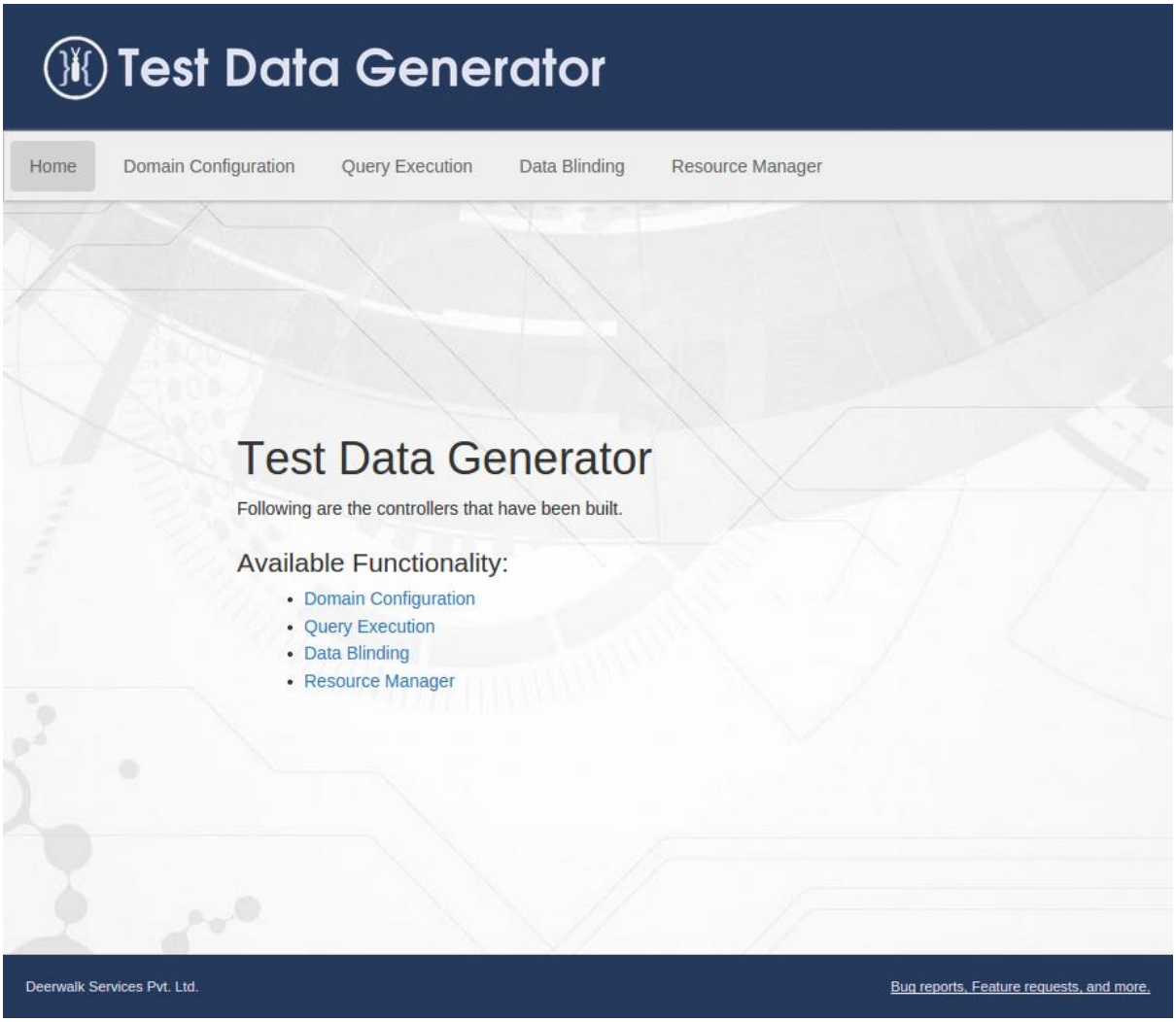
```
grails run-app
```

This will download the remaining dependencies and start up the server. When complete, notice that the app runs on port `8090` and not the conventional `8080`; this is to avoid conflicts if a server is already running. If port `8090` is occupied as well, either kill the other app, or change the server port to something else by modifying `grails.server.port.http` in `BuildConfig.groovy`.

When the server is started, navigate to the following url on any computer in the same network,

```
http://<server-system-ip>:8090/TestDataGenerator101/
```

You should be greeted with the homepage.



Now you can either use the navigation menu, or the links to navigate to the desired page.

The basic flow of test data generator goes like this,

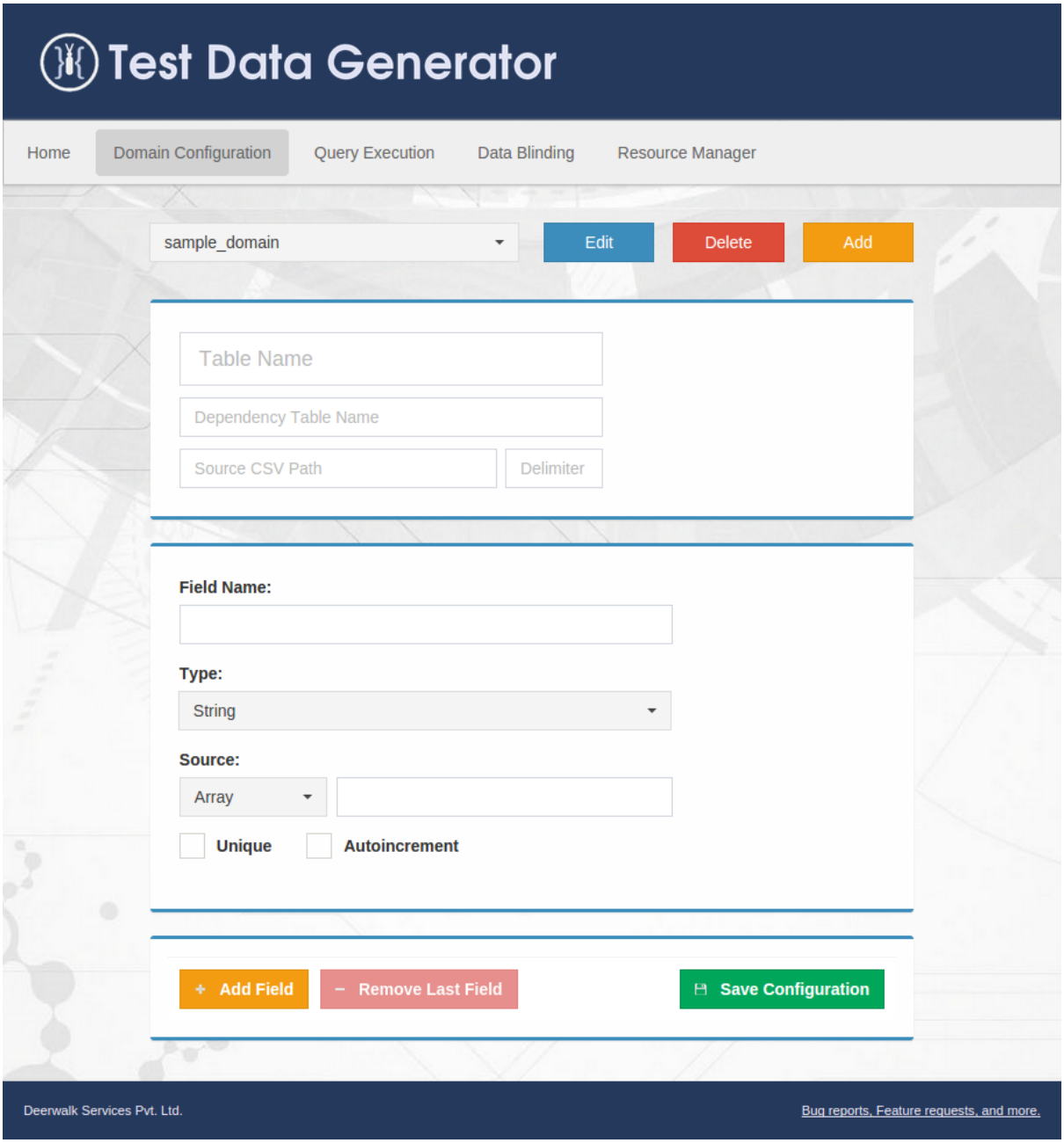
- Create a domain according to your requirements.
- Select the domain for query executions.
- Run the query and retrieve the generated test data.

Side functionalities include,

- Editing existing domains.
- Blinding existing data by providing the logic.
- Managing the resources for the web app. This includes the input files as data, the output files, domain configuration files etc.

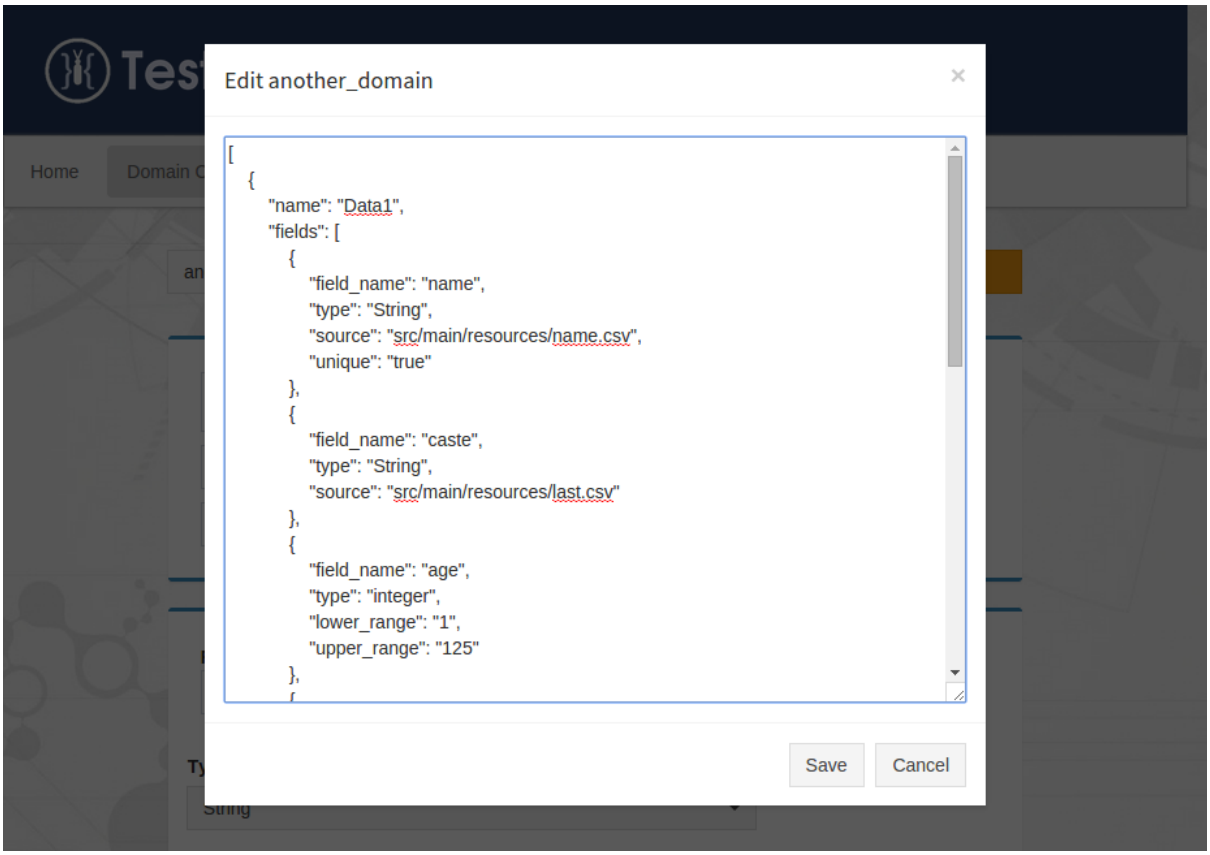
Domain Configuration

To create a new domain, navigate to the **Domain Configuration** tab.



If you are familiar with the domain structure, nothing in this page should come as a surprise. You can add domains, delete them, add tables to a domain, and add fields to the tables. Two domain examples have been provided as samples for you to view. You can delete them safely if you have no use for it. Please refer to the section on domain structure to get the details about what each field in this page means and does.

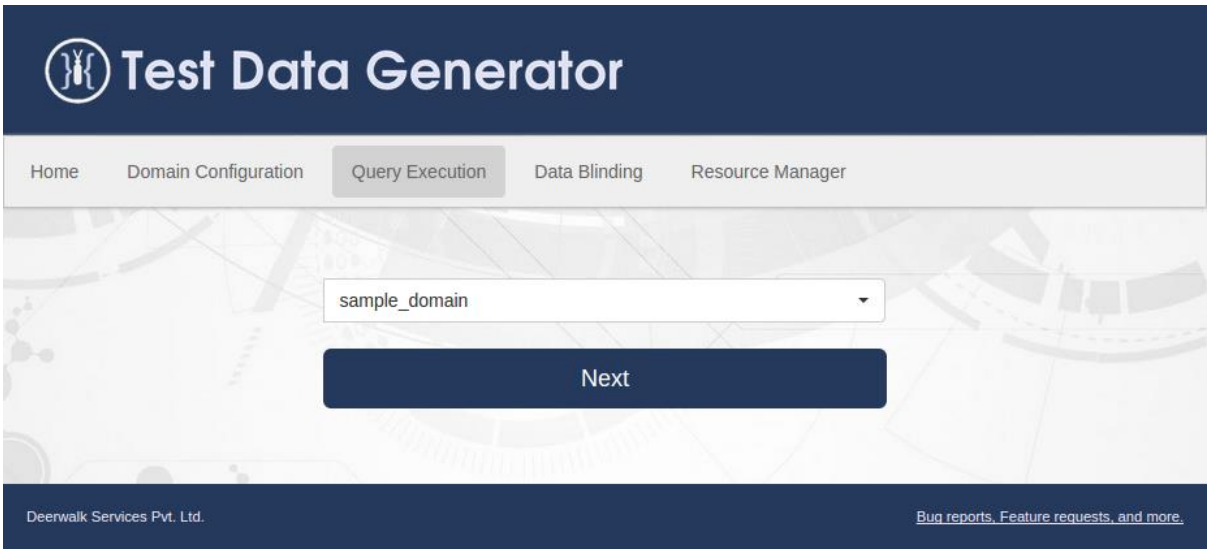
If you need very fine control over your domain, or if you want to modify something in an existing domain, you can use the edit domain functionality.



As you can see, the domains are nothing more than regular `JSON` files.

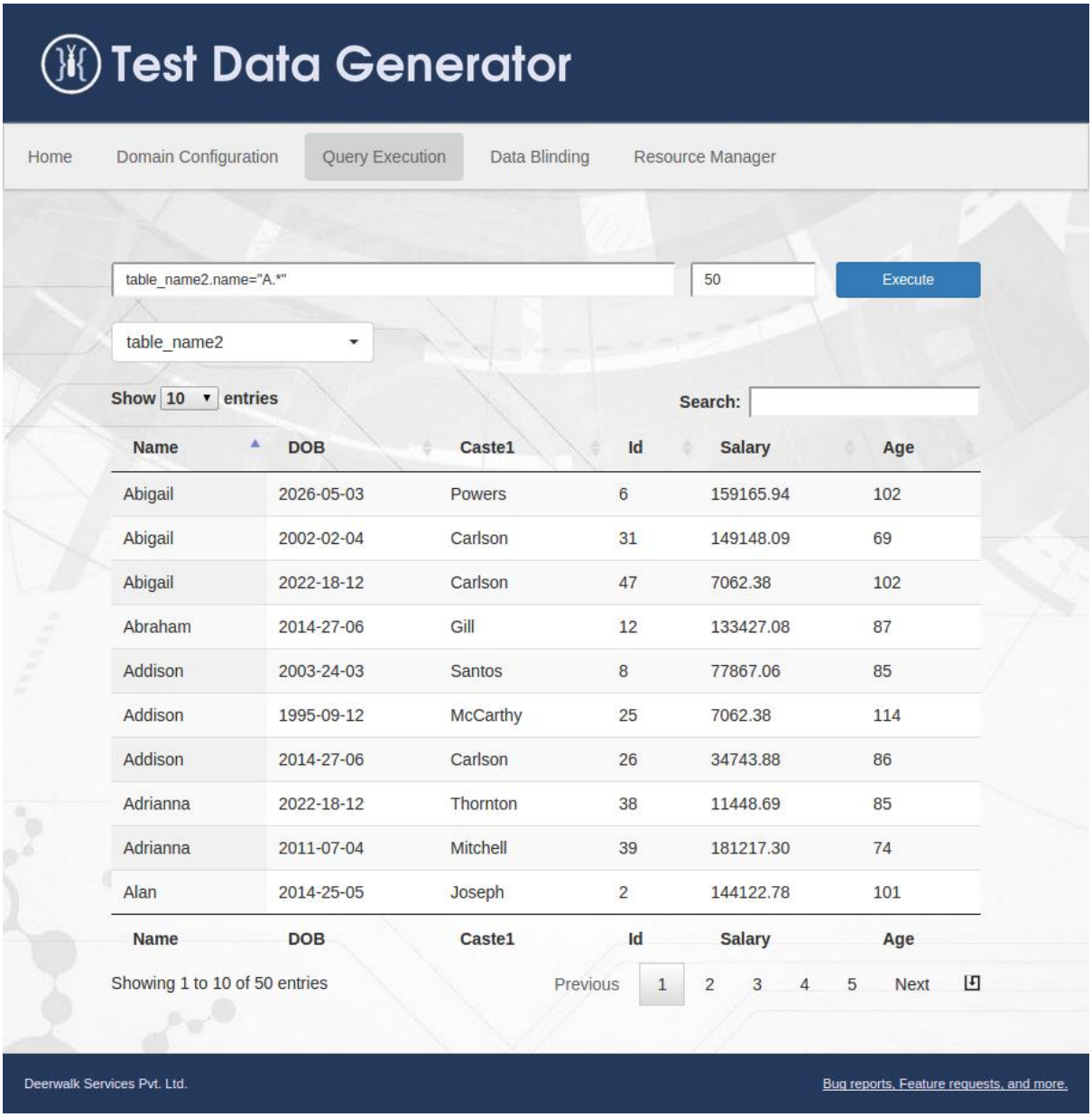
Query Execution

With your domains prepared, you can run queries in them. Let's move to the **Query Execution** tab.



As you would expect, you'll need to choose the domain first. All the domain files in `domain_config` folder are displayed.

In the next page, you'll enter your query and number of test data records you desire.

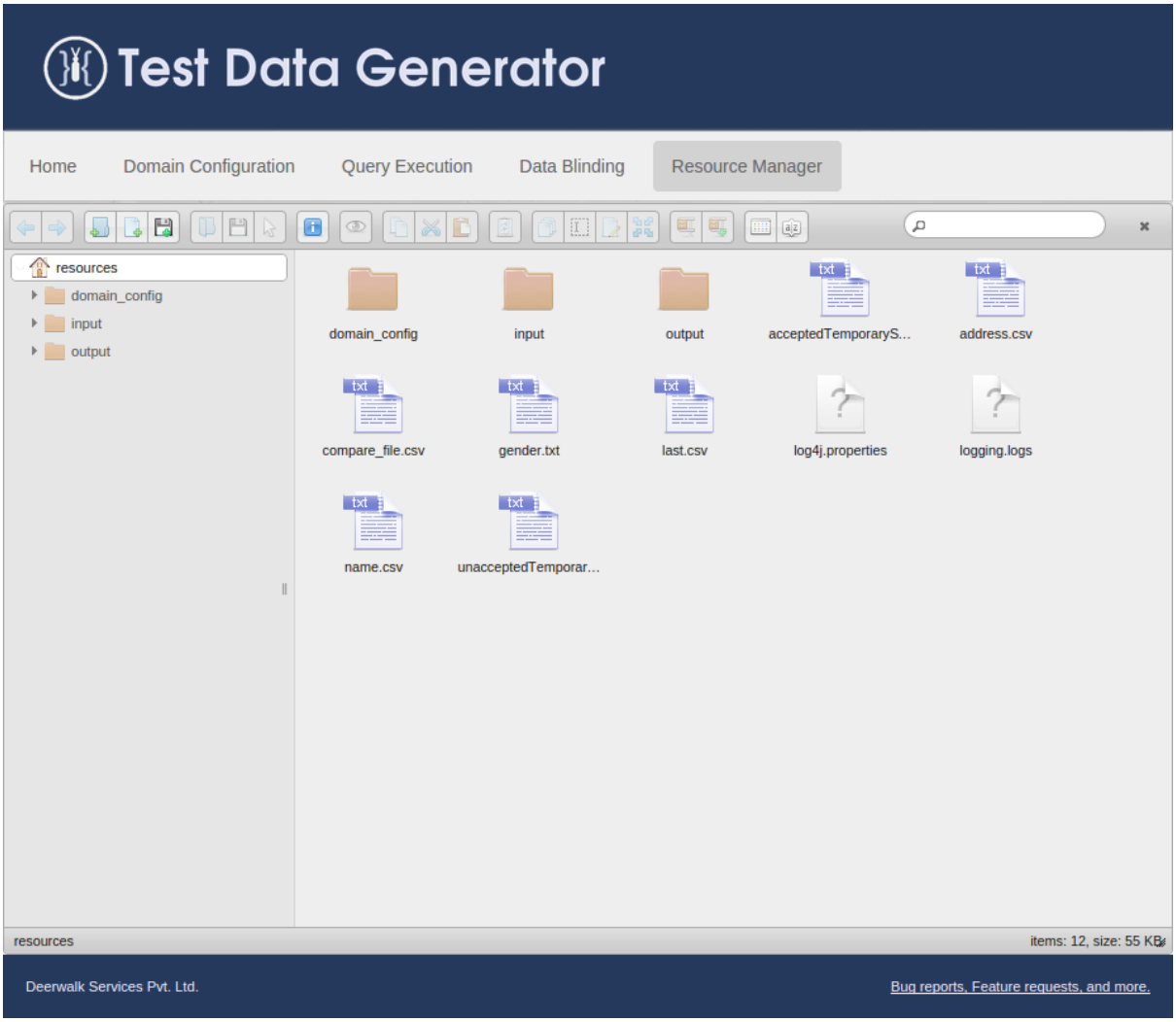


You have functionalities for switching between generated tables, searching records, ordering, and downloading the output files as `csv`. The `Download button` is right next to the `Next button`, near the bottom right.

To know more about the queries you can run and their format, please refer to the section on query structure.

Now that the core features of the product have been explored, let's see the side ones.

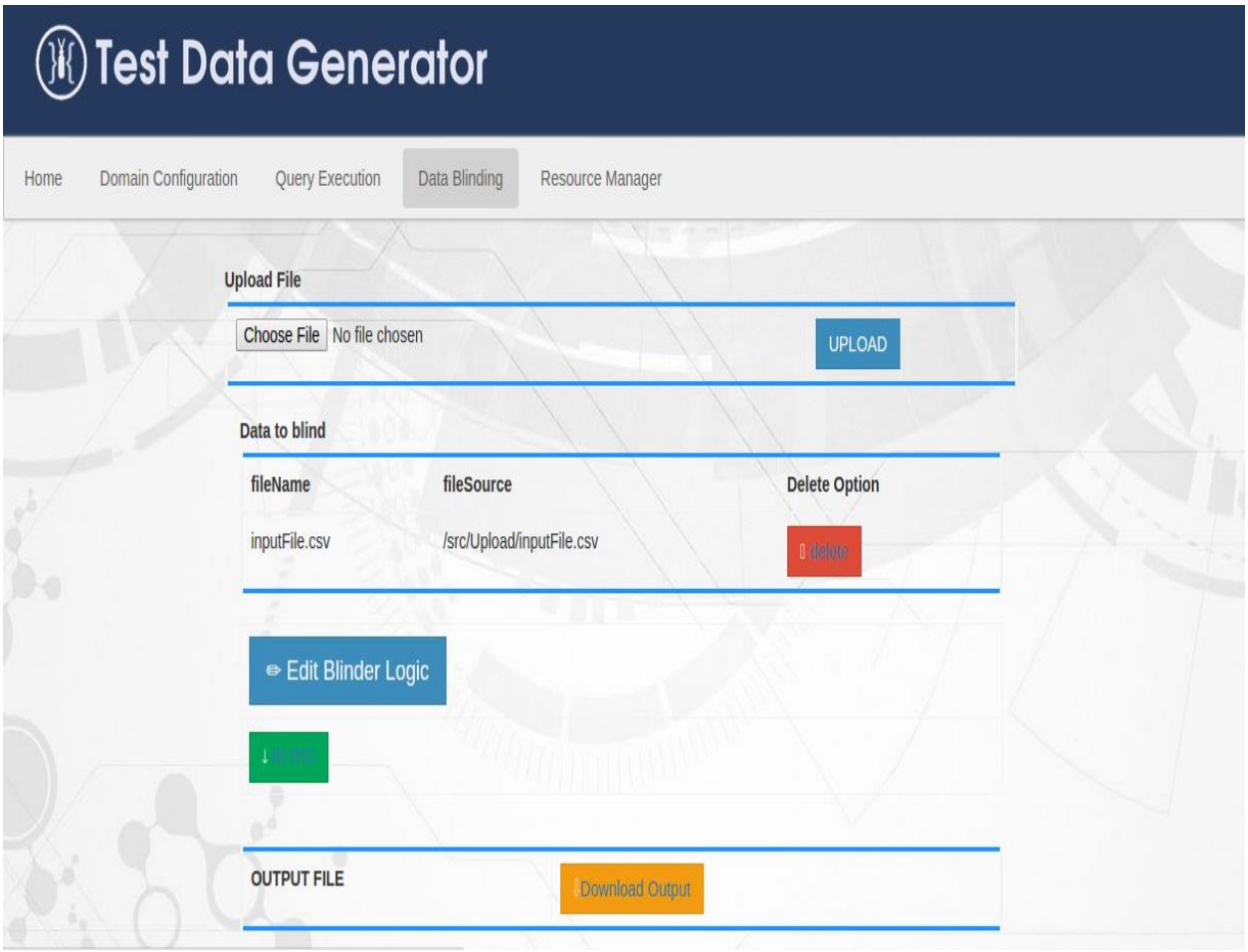
Resource Manager



You are given full access to the resources directory in the project folder. You can add files, create directories, move them, delete them and more. As briefly mentioned earlier, the resources include the domain configuration files, the input files and the outputs generated. The three folders should not be removed under any circumstance, as this will create many critical issues with the program. Also, please do not put anything of importance in the output directory, its contents are deleted before every query execution.

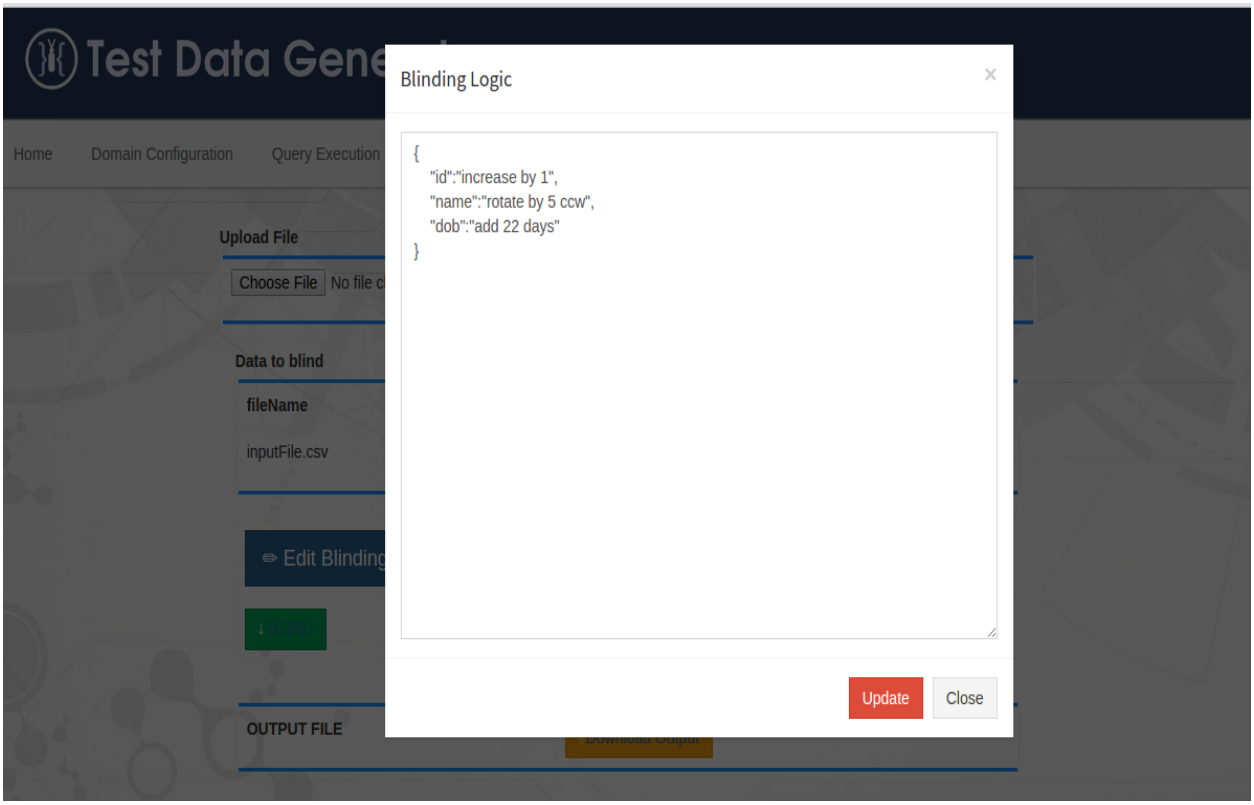
Data Blinder

This page is different from the other pages especially in functionality and view as well. As per the word says, it blinds the data provided by user in certain format. That format should be provided by the user in `JSON` file.



The top most **Upload File** label indicates the section from where we upload the data that is to be blinded. While that uploaded data resides at **Data to blind** section, which can be deleted. After that hassle begins with the **Edit Blinder Logic** button, where you have to edit your blinder logic. And make sure, before you jump to the **Download Output** button to download the file with desired output, green color **BLIND** button is waiting for you to perform the intended blinding task.

The following image shows blinding logic applied. Please refer to the section on Data Blinding Rules to learn more about these logic.



Limitations of Blinding Frontend:

- Not more than one file uploading at the same time.
- Editing of the Blinding Logic should be done at real time in the system, no file uploading can be done. Though you can save your Blinding Logic in that file and access in near future.
- Before you download the desired Output file, be sure that you clicked the Blind button. If not you'll counter the previously generated output.