**Contextual Data Generator**

Table of Contents

[Chapter 1 Introduction 3](#_Toc160549496)

[Purpose 3](#_Toc160549497)

[Scope 3](#_Toc160549498)

[Overview 3](#_Toc160549499)

[Contextual-Data Generator Features 4](#_Toc160549500)

[Chapter 2 Architecture 6](#_Toc160549501)

[Architecture 6](#_Toc160549502)

[Declarative Data Language 7](#_Toc160549503)

[Example 10](#_Toc160549504)

[Chapter 3 Implementation 13](#_Toc160549505)

[Introduction 13](#_Toc160549506)

[Data Type supported. 13](#_Toc160549507)

[Data Type proposed. 15](#_Toc160549508)

[Chapter 4 How To Use 16](#_Toc160549509)

[Setup. 16](#_Toc160549510)

[Command Line Interface. 16](#_Toc160549511)

[API based Interface. 17](#_Toc160549512)

[ Create instance of ContextualDataGenerator class. 17](#_Toc160549513)

[ Create instance of GeneratorOutputComponent. 17](#_Toc160549514)

[ Trigger data generation 18](#_Toc160549515)

[ Complete Example 19](#_Toc160549516)

[Chapter 5 What Next 21](#_Toc160549517)

[Road Map 21](#_Toc160549518)

Chapter 1   
Introduction

# Purpose

The purpose of the document is to explain the approach, architecture, features and implementation details of contextual data generation.

# Scope

This document is an architecture document that provides a comprehensive overview of the proposed approach for Contextual Data generation. It presents architectural views to depict different aspects of the system. I also provide quick overview on implementation and feature set available with this.

As data generation is concerned, it focuses first on only one V (variety) as targets to test all data application. You should be able to generate data for all possible applications and in various formats that varies from structured from unstructured data.

In later phases, we will target other V3 like velocity, volume, and veracity.

# Overview

Contextual-Data Generator is to provider user a declarative language to define their data format. It would give user flexibility to define format of data that varies from structured data to unstructured data.

The document covers the following points:

* Architecture Design of Contextual-data generator
* Features of Contextual-Data Generator
* About declarative language for data generation
  + Language constructs
    - Inbuilt Data Types and their syntax
      * Generic Data Types
      * Domain Specific data types
    - Operators
      * Metacharacter
      * Boolean "or"
      * Grouping
      * Quantification
* Examples covering data generation in different format using declarative language.
* Implementation details
* How to use Contextual-Data Generator

# Contextual-Data Generator Features

Contextual-Data generator has following features:

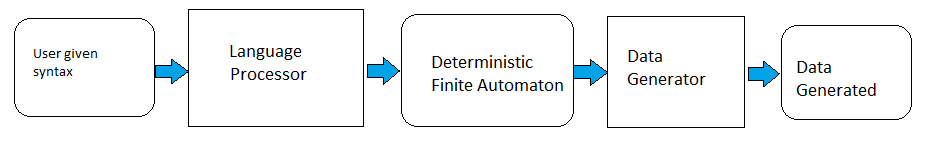
* **Declarative language for data format** 
  + Flexible enough to generate data varies from structured to unstructured. With this you can define format like XML, JSON, CSV, log type data etc.
* **Predefined 30 + inbuilt data types.**
  + Inbuilt data type provides flexibility to generate specific values like mobile number, UUID, IP address, names, country, and city. Etc.
  + You just include data type in input expression. It will generate data embedded with data type value.
  + It has some generic properties that applies to all data types and it also have data type specific properties.
  + Though, parameters come with default value, you can change from default to generate data type value as per your requirement.
* **Wide variety of parameters for each data type to change output data format.**
* **Create your own data type**
  + Though Contextual Data generator provides numerous inbuilt data types for most frequently used values.
* **Inbuilt connector to persisted generated data sources like File, MongoDB, Kafka, Cassandra, MySQL etc.**
  + With inbuilt connectors, data generated with generator will be pushed to source directly. You don’t need to write code for pushing data to any source.
* **Create your own connector to store generated data somewhere else.**
  + We have already in our roadmap to implement connector for some sources like couchDB , Redis, MongoDB etc. we can implement connector for any source by implementing some interface and just specify during data generation, It will generate data for you and persist data as per your connector.
* **Regular Expression based data generation supported.**
  + You wanted a data type value of specific pattern which cannot be generated by existing data type, you can use Regex Based Data type to generate value based on user regular expression.
* **Advanced data generator features**
  + You can specify the percentage of specific values for each data type.
  + Randomly generated values can be constrained to a range of values.
  + Randomly generated values can be constrained to a number of unique values.
* **More Flexibility**
  + Declarative language come with some powerful operators that gives you flexibility to generate data in complicated formats.
    - Boolean operator - also name alternate operation, used provide alternate values at any point of input user expression.
    - Grouping operator - to group of values in single value
    - Quantification operator - to make specific part optional or to generate multiple type
  + Declarative take care of co-relation among different data type used. Will generate data accordingly. Example: if you have used Country and City data type in user expression, values for these data type will be correlated means city value will be based on value random country value generated.
  + **Referencing already defined data type** - you can reference already defined data type by id to generate same value at multiple place in expression.

Chapter 2   
Architecture

# Architecture

Contextual-Data Generator Architecture involves several specialized components that interact with each other to form a highly reliable system. Below is the architecture diagram for the Contextual-Data Generator.

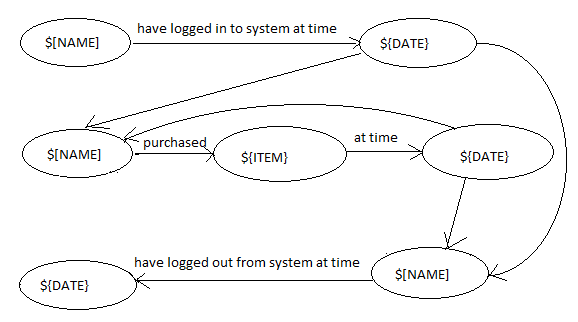
**Language processor** and **data generator** are key processing units and DFA is intermediate data format and



**Figure 1.1: Architecture of Data Generation proposed approach**

* **Language processor**: will translate input (as per declarative data language) provided by user into **non-deterministic finite automaton** ([NFA](http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton)), which will be then made [deterministic](http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton).
* **Data Generator:** will use Deterministic finite automaton (DFA) to generate synthetic data.

Figure 1.2 show NFA of user activity at Online Shopping System.



**Figure 1.2: NFA for User purchase activity at Online Shopping Sytem**

With derived NFA , Following pattern would be generated

**Sanjiv** have logged in to system at time **11.00 AM 21/12/13**

**Sanjiv** puchased **T-Shirt** at time **11.05 AM 21/12/13**

**Sanjiv** puchased **Java Book** at time **11.07 AM 21/12/13**

**Sanjiv** have logged out from system at time **11.10 AM 21/12/13**

**OR**

**Rajiv**  have logged in to system at time **11.00 AM 21/12/13**

**Rajiv** have logged out from system at time **11.02 AM 21/12/13**

# Declarative Data Language

Contextual-Data Generator provides user a declarative data language to define their data format. It would give user flexibility to define format of data that varies from structured data to unstructured data.

Declarative data language consists of Data types and operator symbols that denotes set of strings and operations over these sets, respectively. There can be multiple data language expression that matches a particular set.

In most [formalisms](http://en.wikipedia.org/wiki/Formalism_(mathematics)), if there exists at least one data language expression that matches a particular set then there exists an infinite number of other data language expression that also match it—the specification is not unique.

Most formalisms provide the following operations to construct Declarative Data Language.

* + Inbuilt Data Types and their syntax
  + Language operators
    - Boolean "or"
    - Grouping
    - Quantification
  + Co-relation and referencing among Data Type
* **Inbuilt Data Types and their syntax**

Data Type is a basic building block of Declarative Data Language. Data Type are integral part of Declarative lannguage. Operatiors will be applied to construct various Data formats. And alos you can embed any static data syntax that will remain same with all records generated.

Following table shows proposed list of inbuild data type support in declarative data language

|  |  |  |  |
| --- | --- | --- | --- |
| Blank | Boolean | City | Color |
| Company Name | Country | Credit Card | Credit Card Type |
| Currency | Currency Code | Custom List | Date |
| Domain Name | Email Address | Encrypt | First Name |
| First Name (Female) | First Name (Male) | Formula | Frequency |
| Full Name | Gender | Gender (abbrev) | Given Name |
| GUID | Hex Color | IP Address v4 | IP Address v6 |
| ISBN | Language | Last Name | Latitude |
| Longitude | MAC Address | Money | My List |
| Normal Distribution | Number | Paragraphs | Password |
| Phone | Province | Province (abbrev) | Race |
| Regular Expression | Row Number | Sentences | Sequence |
| Shirt Size | Short Hex Color | SSN | State |
| Street Address | Street Name | Street Number | Street Suffix |
| Suffix | Template | Time | Time Zone |
| Title | Top Level Domain | URL | Username |
| Words | Zip | State (abbrev |  |

*Table 2.1 show proposed list of inbuild data types*

**Note**: User can define their own data in form of **custom list** and **regular expression** data type.

Detail about syntax of each data type supported will be covered in next phase.

**Language operators**

Language provides following set of operator to construct data expression.

* **Boolean "or"**

A vertical **${“type”:”|”}$** separate alternatives.

Example:

**${“name”:”IPV4”}$ ${“type”:”|”}$ ${“name”:”IPV6”}$**

Can match to 192.168.145.93 or FE80:0000:0000:0000:0202:B3FF:FE1E:8329

* **Grouping**

Parentheses **${“type”:”(”}$** and **${“type”:”)”}$** and are used to define the scope and precedence of the [operators](http://en.wikipedia.org/wiki/Operator_(programming)) (among other uses).

* **Quantification**

A quantifier after a data type (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark **${“type”:”?”,”range”:”0-1”}$**, the asterisk **${“type”:”\*”,”range”:”0-5”}$**, (derived from the Kleene star), and the plus sign **${“type”:”+”,”range”:”1-5”}$**,  (Kleene plus).

|  |  |  |
| --- | --- | --- |
| **${“type”:”?”}$** |  | The question mark indicates there is *zero or one* of the preceding element |
| **${“type”:”\*”}$** |  | The asterisk indicates there is *zero or more* of the preceding element. |
| **${“type”:”+”}$** |  | The plus sign indicates there is *one or more* of the preceding element. |

Example:

**${“name”:”IPV4”}$ ${“type”:”(”}$ , ${“name”:”IPV4”}$ ${“type”:”)”}$ ${“type”:”+”}$**

will match to any comma separated list of IPV4 addresses like 192.168.145.1,192.168.145.2,192.168.145.3

**Note**: There are other operators (**Metacharacter,**[*concatenation*](http://en.wikipedia.org/wiki/Concatenation)*,* [*alternation*](http://en.wikipedia.org/wiki/Alternation_(formal_language_theory))*,* [*Kleene star*](http://en.wikipedia.org/wiki/Kleene_star)) that we can take into consideration which would help to construct complicated data language expression.

* **Referencing defined Data Type and Co-relation among Data Type**

**Referencing defined Data Type**

We have a way you can reference by id already defined user data type in input expression and can use them other places in expression to produce same random value.

See the example, below expression have dataType used with id **myName** and later in expression it is referenced.

**${“name”:”NUMBER”,”id”:”myNum”}$ , ${“name”:”BOOLEAN”}$, ${”id”:” myNum”}$**

With given expression, data would be generated like this

**12345,true,12345**

**54675,false,54675**

**Co-relation among Data Type**

There is a way you can define co-relation between among different data type expression by parameter “group”.

See the example below expression made up of two data types COUNTRY and CITY with same group “myG”.

**${“name”:”COUNTRY”,”group”:”myG”}$ , ${“name”:”CITY”,”group”:” myG”}$**

With this, city value will based on random country value generated.

India,Delhi

China, Beijing

# Example

Following table shows few data language expressions and possible generated data.

|  |  |
| --- | --- |
| Language Expression | Generated Data |
| <user>  <name>${“name”:”NAME”}$</name>  <gender>${“name”:”GENDER”}$</gender>  </user> | <user>  <name>Bob Raman</name>  <gender>M</gender>  </user>  <user>  <name>Paul Piaia</name>  <gender>A</gender>  </user> |
| ${“name”:”NAME”}$,${“name”:”MOB”}$,${“name”:”GENDER”}#,${“name”:“ADDRESS”}$ | Bob Raman,9087654325,M,4028 naya bazar delhi  Paul Piaia,9990447339,M,K-24 sectort-11 Noida |
| ${“name”:”NAME”,”id”:”myName”}$ have logged in to system at time ${“name”:”TIME”}$  ${“type”:”(“}$ ${”id”:”myName”}$ purchased ${“name”:”ITEM”}$ at time ${“name”:”TIME”}$ ${“ type”:”)“}$ ${“ type”:”\*“}$  ${”id”:”myName”}$ have logged out from system at time ${“name”:”TIME”}$ | **Sanjiv** have logged in to system at time **11.00 AM 21/12/13**  **Sanjiv** puchased **T-Shirt** at time **11.05 AM 21/12/13**  **Sanjiv** puchased **Java Book** at time **11.07 AM 21/12/13**  **Sanjiv** have logged out from system at time **11.10 AM 21/12/13** |
| ${“ type”:”(“}$ ${“name”:”DATE”}$ ${“name”:”TIME”}$ NAI=${“name”:”DEVICEID”}$@tsp08.sprintpcs.com started session on PDSN=${“name”:”IPV4”}$ at ${“name”:”DATE”}$ ${“name”:”TIME”}$ IP-Type = Mobile; Service-Type = 1xRTT; BSID = 101800000772 and Assigned IP address=${“name”:”IPV4”}$  ${“ type”:”|“}$  ${“name”:”DATE”}$ ${“name”:”TIME”}$ NAI=${“name”:”DEVICEID”}$@tsp08.sprintpcs.com continued session on PDSN=${“name”:”IPV4”}$ for 900 seconds. Data to radio=${“name”:”NUMBER”}$ bytes and data from radio=${“name”:”NUMBER”}$ bytes.  ${“ type”:”|“}$  ${“name”:”DATE”}$ ${“name”:”TIME”}$ NAI=${“name”:”DEVICEID”}$@commandcenter.sprintpcs.com stopped session on PDSN=${“name”:”IPV4”}$ at ${“name”:”DATE”}$ ${“name”:”TIME”}$ after 15 seconds; IP-Type = Mobile; Service-Type = 1xRTT;Data to radio=${“name”:”NUMBER”}$ bytes and data from radio=${“name”:”NUMBER”}$ bytes. | 2012/04/01 00:00:00.772 NAI=**7632813148**@tsp08.sprintpcs.com started session on PDSN=68.28.177.69 at 2012/04/01 00:00:00; IP-Type = Mobile; Service-Type = 1xRTT; BSID = 101800000772 and Assigned IP address=10.151.243.196  2012/04/01 00:00:00.918 NAI=**7632663602**@tsp08.sprintpcs.com continued session on PDSN=68.28.249.85 for 900 seconds.Data to radio=57 bytes and data from radio=13 bytes.  2012/04/01 00:00:00.994 NAI=**7636205247**@tsp08.sprintpcs.com started session on PDSN=68.28.153.69 at 2012/04/01 00:00:00; IP-Type = Mobile; Service-Type = 1xRTT; BSID = 1039000111D1 and Assigned IP address=10.151.200.30  2012/04/01 00:00:00.584 NAI=**7632660437**@tsp08.sprintpcs.com continued session on PDSN=68.28.153.85 for 138600 seconds.Data to radio=30 bytes and data from radio=30 bytes.  2012/04/01 00:00:00.684 NAI=**5338253293**@commandcenter.sprintpcs.com stopped session on PDSN=68.28.57.69 at 2012/04/01 00:00:00 after 15 seconds;IP-Type = Mobile;Service-Type = 1xRTT;Data to radio=**150** bytes and data from radio=**140** bytes. |
| ${“name”:”COMPANY”}$#3,120.00#${“name”:”NUMBER”}$#11,43,210#35,757.89#3,145.00#3,172.95#3,107.75#3,158.90#${“name”:”DATE”}$ | **INFY**#3,120.00#**1.23**#11,43,210#35,757.89#3,145.00#3,172.95#3,107.75#3,158.90#30-May-13  **DLF**#3,120.00#**3.10**#11,43,210#35,757.89#3,145.00#3,172.95#3,107.75#3,158.90#30-May-13  **HINDUNILVR**#3,120.00**#5.56**#11,43,210#35,757.89#3,145.00#3,172.95#3,107.75#3,158.90#30-May-13  **PNB**#3,120.00#**6.89**#11,43,210#35,757.89#3,145.00#3,172.95#3,107.75#3,158.90#30-May-13  **POWERGRID**#3,120.00**#7.89**#11,43,210#35,757.89#3,145.00#3,172.95#3,107.75#3,158.90#30-May-13 |

*Table 2.2: Examples*

You can construct any data language expression based on data type and operators available to generate data in any format.

Chapter 3   
Implementation

# Introduction

Below is architecture of solution, where **Language processor and data generator** are key processing units and DFA is intermediate data format and

Implementation approach is given below:

**Step 1: Convert User expression to NDFA**

Thompson’s Construction Algorithm(**CTA**) have been implemented in java and been used for user expression embedded with data types to NDFA conversion

**Step 2: Convert NDFA to DFA**

Then generated NDFA out of CTA converted into DFA.

**Step 3: Generate random data out of the DFA**

Once we have DFA of user expression embedded with data types, it is used to generated random patterns and data type place holder will be replace by random value of that type.

# Data Type supported.

Table shows some of data types currently supported. Table talks about data Type, syntax, parameter supported, and sample use and sample data generated.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Description - | Syntax - Example | Sample generated data |
| DATE |  | ${  "name":"DATE",  "format":"yyyy-MM-dd HH-mm-ss",  "from" : "21/12/2012" ,  "to" : "21/12/2014"  }$ |  |
| BIGNUMBER |  | ${  "name":"BIGNUMBER",  "size" : "20"  }$ |  |
| ALPHANUMERIC |  | ${  "name":"ALPHANUMERIC",  "size" : "32"  }$ |  |
| BOOLEAN |  | ${  "name":"BOOLEAN",  "abbrev" : "true" ,  "uppar" : "true"  }$ |  |
| COLOR |  | ${"name":"COLOR"}$ |  |
| COMPANY |  | ${"name":"COMPANY"}$ |  |
| COUNTRY |  | ${  "name":"COUNTRY",  "isCapital":"true",  "isCode":"false"  }$ |  |
| CURRENCY |  | ${"name":"CURRENCY"}$ |  |
| CURRENCYCODE |  | ${"name":"CURRENCYCODE"}$ |  |
| FREQUENCY |  | ${"name":"FREQUENCY"}$ |  |
| GENDER |  | ${  "name":"GENDER",  "abbrev" : "true" ,  "uppar" : "true"  }$ |  |
| GUID |  | ${"name":"GUID"}$ |  |
| HEXCOLOR |  | ${"name":"HEXCOLOR"}$ |  |
| DIPV4 |  | ${  "name":"DIPV4",  "format":"251.\*.\*.\*"  }$ |  |
| DIPV6 |  | ${"name":"DIPV6"}$ |  |
| LATITUDE |  | ${  "name":"LATITUDE",  "format":"#.##########"  }$ |  |
| LONGITUDE |  | ${  "name":"LONGITUDE",  "format":"#.##########"  }$ |  |
| MACADDRESS |  | ${  "name":"MACADDRESS",  "seperator":":",  "isCap":"false"  }$ |  |
| ROWNUMBER |  | ${  "name":"ROWNUMBER",  "decimal":"4",  "start" : "1" ,  "end" : "1000"  }$ |  |
| TIME |  | ${"name":"TIME" , "format":"true" , "from" : "10:45 AM" , "to" : "11:45 PM" }$ |  |
| TIMEZONE |  | ${"name":"TIMEZONE"}$ |  |
| TITLE |  | ${"name":"TITLE"}$ |  |
| MONEY |  | ${  "name":"MONEY",  "decimal":"2" ,  "currencySymbol" : "Random",  "start": "1" ,  "end" : "100"  }$ |  |
| REGULAREXPRES |  | ${  "name":"REGULAREXPRESSION",  "regex":"d{2}-d{2,4}"  }$ |  |
| CUSTOMLIST |  | ${"name":"CUSTOMLIST", "values" : "sanjiv|rajiv|nikki"}$ |  |
| TOPDOMAINNAME |  | ${"name":"TOPDOMAINNAME"}$ |  |
| DOMAINNAME |  | ${"name":"DOMAINNAME"}$ |  |
| EMAIL |  | ${"name":"EMAIL"}$ |  |
| URL |  | ${  "name":"URL",  "protocal":"false" ,  "host" : "true" ,  "path" : "true" ,  "queryString" : "false"  }$ |  |

**Table 3.1 Inbuilt data types supported, their parameter and use**

# Data Type proposed.

Following table shows proposed list data types that are not implemented. As these type of data frequently required in applications, will implement and integrate with contextual-data generator on requirement basis.

|  |  |  |  |
| --- | --- | --- | --- |
| Credit Card | City | Credit Card Type | Encrypt |
| First Name | First Name (Female) | First Name (Male) | Formula |
| Frequency | Full Name | Given Name | ISBN |
| Last Name | My List | Normal Distribution | Paragraphs |
| Password | Phone | Province | Province (abbrev) |
| Race | Sentences | Sequence | Shirt Size |
| SSN | State | State (abbrev) | Street Address |
| Street Name | Street Number | Street Suffix | Template |
| Username | Words | Zip |  |

**Table 3.2: proposed list data types that are not implemented.**

Chapter 4   
How To Use

Using Contextual Data generator is very simple and straight forward. There are two options available.

* Command line Interface.
* API based Interface.

# Setup.

Copy configuration file from **src/main/resources** to **/var/data-generator/conf**

* Config.properties
* dataTypeClasses.properties
* Messages.properties
* log.properties

Change **src/main/resources/ System.properties** accordingly

messages.file.path=/var/data-generator/conf/Messages.properties

configs.file.path=/var/data-generator/conf/Config.properties

logging.file.path=/var/data-generator/conf/log.properties

datatype.classes.file.path=/var/data-generator/conf/dataTypeClasses.properties

# Command Line Interface.

Moved to directory where ContextualDataGenerator.jar jar.

* For generating data in file, use:

**java -jar ContextualDataGenerator.jar file --numberofrecords 1000 --inputDir /home/sanjivsingh/BDDG/input/ --outputDir /home/sanjivsingh/BDDG/output/  --seperator ""**

First parameter **file** is always type of connector where you want the dump the generated data.

**--numberofrecords <Number of records>**

**--inputDir <input directory where user expression resides>**

**--outputDir** **<output directory where generated file would be created>**

**--seperator <string value to separate records in file, by default new line>**

* For generating data in rabbitMQ queue , use :

**java -jar ContextualDataGenerator.jar rabbitmq --numberofrecords 1000 --inputDir /home/sanjivsingh/BDDG/input/ --host 192.168.145.53  --port 3009 --queue myQueue**

First parameter **rabbitmq** is always type of connector where you want the dump the generated data.

**--numberofrecords <Number of records>**

**--inputDir <input directory where user expression resides>**

**--host <server ip of rabbitmq server>**

**--port <port of rabbitmq server>**

**--queue <queue name>**

# API based Interface.

Following describes how to use Contextual-Data Generator as API.

# Create instance of ContextualDataGenerator class.

String inputDir = "/home/sanjivsingh/BDDG/input/";

// create data generator instance

ContextualDataGenerator bdGenerator = new ContextualDataGenerator(inputDir);

* public **ContextualDataGenerator(String inputDir)**

Constructor which directly instantiates the **ContextualDataGenerator** containing the model contents.

**Parameters:**

**inputDir** - The String parameter is path user input directory where it expects following files.

* + **header** - header content
  + **userInput** - user format of records
  + **footer** - footer content

**header** and **footer** are optional , will be used in case of FileOutput.

# Create instance of GeneratorOutputComponent.

**GeneratorOutputComponent** is an interface which defines contract for output component and Class implementing this interface, will provide implementation of generated records would pushed after generation.

We have currently following implementation.

* + **FileOutputComponent** - Class stores generated records in format for file at disk. It takes parameters
    - **String** **inputDir** : directory path of header and footer files
    - **String** **outputFile** : directory path where generated file will be stored
    - **String** **separator**: any expression will be delimiting records in file , common example is to have separator like Comma “,”or NewLine “\n”.

String inputDir = "/home/sanjivsingh/BDDG/input/";

String outputDir = "/home/sanjivsingh/BDDG/output/";

String seperator = "\n";

GeneratorOutputComponent foc = new FileOutputComponent(inputDir,

outputDir, seperator);

* + **ListOutputComponent** - Class populate in-memory list with randomly generated records that user can use programmatically anywhere for processing / persisting.

GeneratorOutputComponent foc = **new** ListOutputComponent();

* + **RabbitMqOutputComponent** - Class pushes generated records in rabbitMq queue. It takes parameters.
    - **String queueIp** : IP of RabbitMq Server
    - **int port** : port of rabbitMq on which services listening to user connection.
    - **String queueName**: name of the queue in which you want to push generated records.

String queueIp = "192.168.145.53";

**int** port = 3099;

String queueName = "myQueue";

GeneratorOutputComponent foc = **new** RabbitMqOutputComponent(queueIp,

port, queueName);

# Trigger data generation

* + **public void generateData(final int numOfRecords,**

**GeneratorOutputComponent genOutComp)**

Method generates records and store record as per **GeneratorOutputComponent** implementation provided as argument.

**Parameters**:

* + **Int numOfRecords** – Number of records to be generated
  + **GeneratorOutputComponent genOutComp** - instance of class implementing interface GeneratorOutputComponent.

**int** numberOfRecords = 100;

bdGenerator.generateData(numberOfRecords, foc);

**Note**:

* You can have custom implementation by implementing interface GeneratorOutputComponent like for Writing generated data to HDFS , Cassandra or other NoSQL , RDBMS system etc.
* Design is pluggable. You can plugin any GeneratorOutputComponent implementation to alter data generation and to change store anywhere or add few processing login before it will get consumed actually.

# Complete Example

* + - **Generate and store in file**

// create data generator instance

**final** String inputDir = "/home/sanjivsingh/BDDG/input/";

ContextualDataGenerator bdGenerator = **new** ContextualDataGenerator(inputDir);

// create instance of FileOutputComponent

**final** String outputDir = "/home/sanjivsingh/BDDG/output/";

String seperator = "\n";

GeneratorOutputComponent foc = **new** FileOutputComponent(inputDir,

outputDir, seperator);

// generate data and save in file

**int** numberOfRecords = 100;

bdGenerator.generateData(numberOfRecords, foc);

* + - **Populate in-memory list**

// create data generator instance

**final** String inputDir = "/home/sanjivsingh/BDDG/input/";

ContextualDataGenerator bdGenerator = **new** ContextualDataGenerator(inputDir);

// create instance of ListOutputComponent

GeneratorOutputComponent foc = **new** ListOutputComponent();

// generate data and save in file

**int** numberOfRecords = 100;

bdGenerator.generateData(numberOfRecords, foc);

// fetch generated records in-memory and process them

**if** (**null** != foc2.getRecords()) {

System.out.println("Data Generated data :");

**for** (String data : foc2.getRecords()) {

System.out.println(data);

}

}

* + - **Generate and push records into rabbitMQ queue.**

// create data generator instance

String inputDir = "/home/sanjivsingh/BDDG/input/";

ContextualDataGenerator bdGenerator = **new** ContextualDataGenerator(inputDir);

// create instance of RabbitMqOutputComponent

String queueIp = "192.168.145.53";

int port = 3009;

String queueName = "myQueue";

GeneratorOutputComponent foc = new RabbitMqOutputComponent(queueIp,

port, queueName);

// generate data and save in file

**int** numberOfRecords = 100;

bdGenerator.generateData(numberOfRecords, foc);

Chapter 5   
What Next

# Road Map

Following features are in pipeline.

* **Implement and integrate following proposed list data type to Data Generator**
  + City
  + Credit Card #
  + Credit Card Type
  + Encrypt
  + First Name
  + First Name (Female)
  + First Name (Male)
  + Formula
  + Frequency
  + Full Name
  + Given Name
  + ISBN
  + Last Name
  + My List
  + Normal Distribution
  + Paragraphs
  + Password
  + Phone
  + Province
  + Province (abbrev)
* **Implement custom plugins for various output component.**
  + Currently we have implemented plugins for RabbitMq and File System. With that generated data can be feeded to rabbitMQ and stored in form of file on disk.
  + We will be implementing plugins for RDBMS system, NoSQL databases (Cassandra, MongoDB etc.) and Queueing system and HDFS etc.
* **Implement GUI based interface.**
  + To be able to design data expression through web-based UI.
  + To be able to preview or validate data sample generated before generating mass data and feeding into some application as input data.
* **Integrating web resources for row data**

Objective of integrating web resources to data generator is to get real data dump and use for data generation.

We already using Lorem Ipsum - <http://www.lipsum.com/> for random words and sentences.

Following are initial list of web resources

* + **Wikipedia** <https://www.wikipedia.org/>
  + **Country wise Geo Info** - http://download.geonames.org/export/zip/
  + <http://earth-info.nga.mil/gns/html/namefiles.htm#I>