CONFLUENT

JR : Quality Random Data from the Command line

Ugo Landini - Solutions Engineer Manager
Stefano Linguerri - Senior Solutions Engineer

Last updated: 15/07/24

> whoami

```
apiVersion: confluent/v1
kind: senior solutions engineer
metadata:
  name: ugo landini
  nick: ugol
  email: ugo@confluent.io, ugo.landini@gmail.com
  namespace: confluent
  annotations: apache/committer, oss lover, distributed geek
  site: https://ugol.io
  labels:
    family: dad of two
    prev_companies: sun microsystems, sourcesense, vmware,
red hat
  spec:
    replicas: 1
    containers:
      - image: github.com/ugol:latest
```

> whoami

```
apiVersion: confluent/v1
kind: senior solutions engineer
metadata:
  name: stefano linguerri
  nick: eljeko
  email: slinguerri@confluent.io stefano.linguerri@gmail.com
  namespace: confluent
  annotations:
  site:
  labels:
    family:
    prev_companies: sourcesense, red hat
  spec:
    replicas: 1
    containers:
      - image: github.com/eljeko:latest
```

> apropos jr

> apropos jr

- **J**son **R**andom generator
- **J**ust another **R**andom generator
- Similar to **JQ**, which is one of the most used tools https://stedolan.github.io/jq/
- ...

> apropos jr

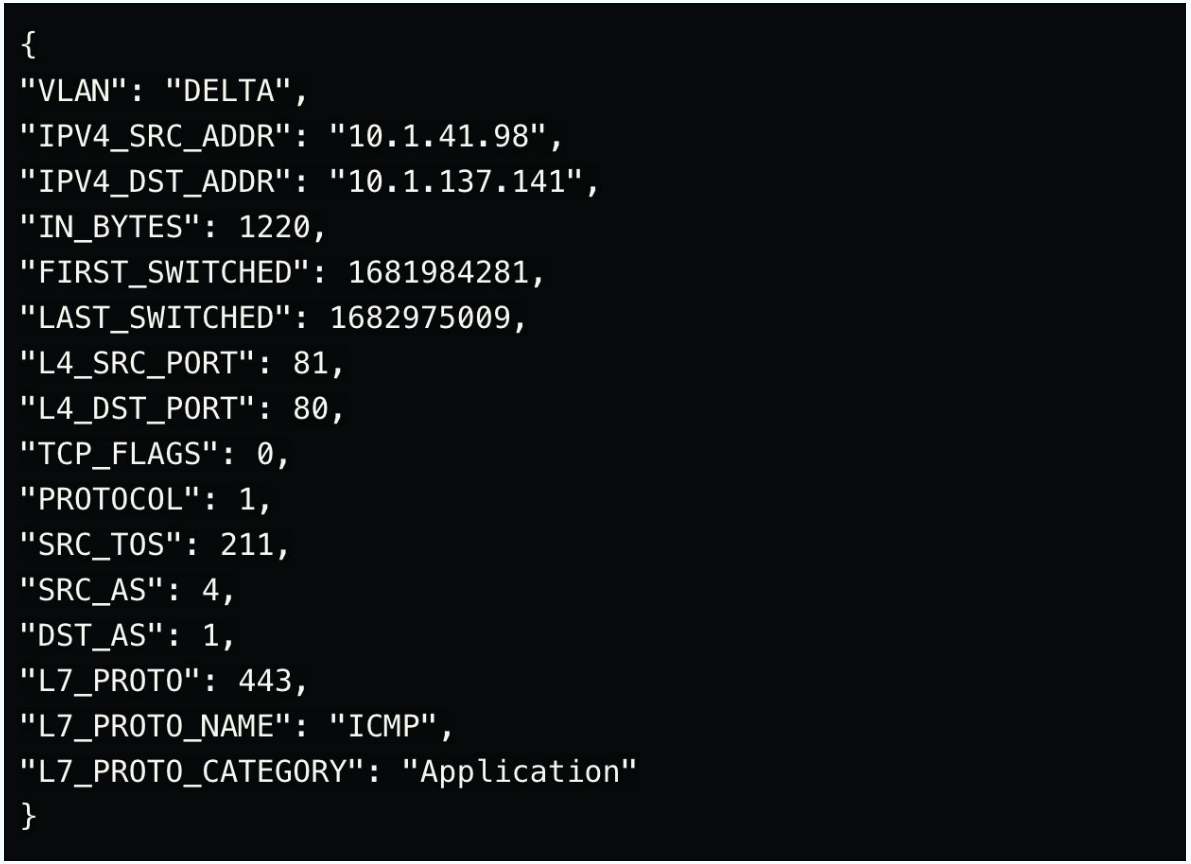
- **J**son **R**andom g
- **J**ust another **R**
- Similar to **JQ**, w
  https://stedolar
- ...

JR

> history | grep jr

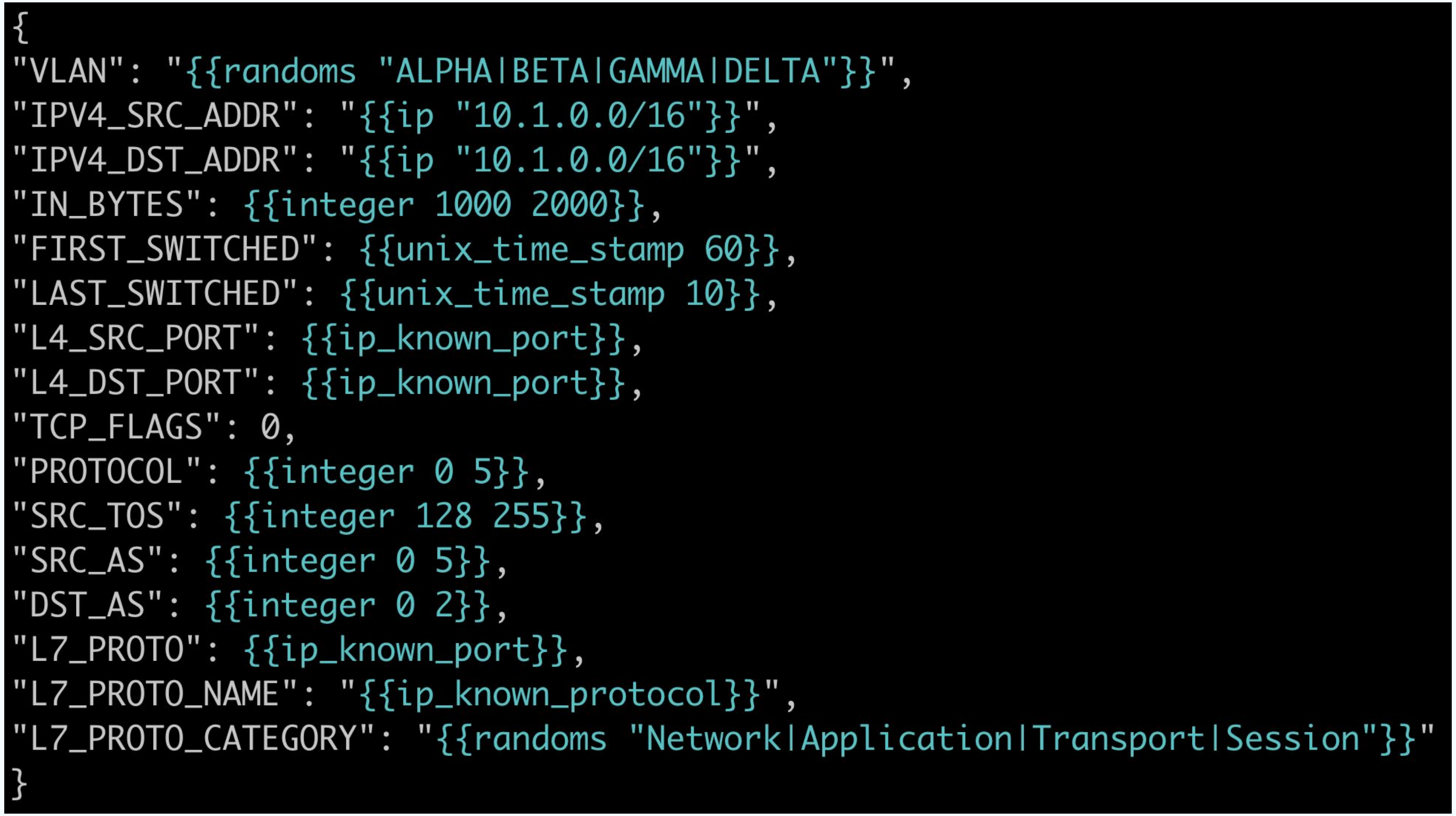
- Had to generate traffic for a customer, on the fly, with just an **example** of a json
- They asked how much this stuff would be **compressed** by the producer, which obviously varies with:
  - different **algorithms**
  - different **throughput**
  - different **batching** kafka configuration
  - can’t use a single json to do that, would be compressed **too much**
- Existing tools couldn’t easily answer this question, and for sure not in a 5 minutes time frame, for example:
  - **Datagen** with custom objects is complex to setup
  - Managed **Datagen** on Confluent Cloud can’t use custom objects and can’t compress

> history | grep jr



```
{
"VLAN": "DELTA",
"IPV4_SRC_ADDR": "10.1.41.98",
"IPV4_DST_ADDR": "10.1.137.141",
"IN_BYTES": 1220,
"FIRST_SWITCHED": 1681984281,
"LAST_SWITCHED": 1682975009,
"L4_SRC_PORT": 81,
"L4_DST_PORT": 80,
"TCP_FLAGS": 0,
"PROTOCOL": 1,
"SRC_TOS": 211,
"SRC_AS": 4,
"DST_AS": 1,
"L7_PROTO": 443,
"L7_PROTO_NAME": "ICMP",
"L7_PROTO_CATEGORY": "Application"
}
```

> history | grep jr



```
{
"VLAN": "{{randoms "ALPHA|BETA|GAMMA|DELTA"}}",
"IPV4_SRC_ADDR": "{{ip "10.1.0.0/16"}}",
"IPV4_DST_ADDR": "{{ip "10.1.0.0/16"}}",
"IN_BYTES": {{integer 1000 2000}},
"FIRST_SWITCHED": {{unix_time_stamp 60}},
"LAST_SWITCHED": {{unix_time_stamp 10}},
"L4_SRC_PORT": {{ip_known_port}},
"L4_DST_PORT": {{ip_known_port}},
"TCP_FLAGS": 0,
"PROTOCOL": {{integer 0 5}},
"SRC_TOS": {{integer 128 255}},
"SRC_AS": {{integer 0 5}},
"DST_AS": {{integer 0 2}},
"L7_PROTO": {{ip_known_port}},
"L7_PROTO_NAME": "{{ip_known_protocol}}",
"L7_PROTO_CATEGORY": "{{randoms "Network|Application|Transport|Session"}}"
}
```

> whois jr

- Is a **template** system, leveraging wonderful Golang **text/template** package
- Has a **CLI** but also **REST APIs**
- Can generate **anything** you could write a template for (so, not tied to json)
- Embeds a specialized **fake** library (no use of existing faking libraries)
- Has **automatic integrity** for related fields (city, zip, mobile, phone, email/company, etc)
- Can maintain **integrity** between objects generated (**relations**)
- It’s been designed for **Kafka**, but can directly output to **Elastic**, **Redis**, **MongoDB, S3, GCS**
- Can talk to **Confluent Schema Registry** for Kafka, serializing in **Avro/Json Schema**

> man jr

- You choose your **template** from the available templates
- You choose **-n** number of objects to generate at each pass
- You choose **-f** frequency
- You choose **-d** duration

```
jr template list
jr template run net_device | jq
jr template run -n 2 net_device | jq
jr template run -n 2 -f 100ms net_device | jq
jr template run -n 2 -f 100ms -d 5s net_device | jq
```

> cat cli

- You have 3 resources: **emitters**, **templates** and **functions**
  - You can list, show and run **templates**
  - You can list available **functions** and test directly (**--run**) without writing a template. There are **131** functions at the moment, and growing
  - **Emitters** are a new concept: you configure different emitters all at once, with different frequency and other parameters, and then you just list/show/run the emitters with a single command

jr function **list** -c finance
jr function **list** card --run
jr function **list** regex --run
jr emitter **list**
jr emitter **run** --dryrun

> man template

- There are **3** different templates to control jr
    - **Key** template, which defaults to **null**
    - **Output** template, which defaults to **Value** only: **{{.V/n}}**
    - **Value** template, which you control in two different ways
        - Embedding directly in the command line (**--embedded**)
        - By name (**user**,**net_device**, etc) for the OOTB templates

```
jr template list
jr template show net_device
jr template show user
jr template run --key '{{key "ID" 100}}' user
jr template run --key '{{key "ID" 100}}' --outputTemplate '{{.K}} {{.V}}' net_device
jr template run --key '{{key "ID" 100}}' --embedded '{{name}} {{email}}' --kcat
```

> man functions

- There are **131** functions at the moment, categorized as
  - People
  - Text utilities
  - Network
  - Context
  - Address
  - Finance
  - Math
  - Phone

```
cat .jr/templates/data/it/movie
jr template run --embedded '{{from "movie"}}'
jr template run --locale IT --embedded '{{from_n "beer" 3}}'
jr template run --locale IT --embedded '{{from_n "actor" 15}}'
jr template run --locale EN --embedded '{{from_n "actor" 15}}'
```

> cat automatic_integrity

- Some functions are “smart”, for example:
  - **Mobile** phones are generated by “inverse” regular expressions, using mobile company numbers valid for the chosen country (**--locale**)
  - Streets, cities, zip codes, phone prefix and more are all **localizable** and **coherent** without doing anything special
  - your **work email** is generated automatically using - if already in the template - previously generated **name**, **surname** and **company**

```
jr template run --embedded '{{name}} {{email}}'
jr template run --embedded '{{name}} {{surname}} {{company}} {{email_work}}'
jr template run user | jq
jr --locale IT template run user | jq
jr --locale FR template run user | jq
```

> echo “hello” 2>&1 >> $LOG

- You can choose different **output** for jr:
    - **stdout** (default)
    - **kafka**
    - **redis**
    - **mongo, mongo atlas**
    - **elastic**
    - **S3**, **gcs**
- Each **output** needs a specific configuration
- Output can easily be extended implementing **Producer** interface

```
jr template run user -o kafka
jr template run user -o kafka -t topic_user -a
jr template run user -o mongo
```

> select * from customers where custID='X1001';

- **Relational Integrity** is where most of similar tools fall. To generate “related” data, they end up having long lists of prebuilt json documents, not at all random. Basically they become equivalent to:
  - **kcat** -P -b localhost:9092 -t topic -K: -l **prebuilt_json.txt**
- jr has two features to help with integrity
  - **preload** to create a bunch of events at the beginning
  - context functions, especially **add_v_to_list, random_n_v_from_list** and **random_v_from_list**

> select * from customers where custID='X1001';

- With preload and context you can for example:
  - generate **1000** random products all at once to a topic
  - generate **100** random customers all at once and then add **1** customer every minute
  - stream **5** random orders every **100ms** by **existing** customers with **existing** products
- To test your streaming apps (**KStream**, **ksqlDB**, **Flink**), you definitely need relations!

jr function **list** -c context
jr template **show** shoestore_shoe
jr template **show** shoestore_customer
jr template **show** shoestore_order
jr template **show** shoestore_clickstream

> select * from customers where custID='X1001';

- Everything in JR is an emitter, also when you run a simple embedded template
- But the real power of emitters is when you run several different ones with different configurations, all at once
  - **jr** emitter **run**
  - This command runs **all** the emitters you configured

jr emitter **show** shoestore
jr emitter **run** shoestore --dryrun
jr emitter **run** --dryrun
jr emitter **run**

> more | grep future

- We need your help!
  - Close issues if you can: https://github.com/ugol/jr/issues
  - **Localizations** in different languages
  - Useful new **functions** for templates
  - Useful pre-configured **emitters** for complex use cases
  - New **output** Producers (every k/v store is a candidate)
- Pls **star, watch** and **fork** the project on Github!
  - ~~The **brew** guys told us that we need a minimum of:~~
  - ~~**30** forks~~
  - ~~**30** watchers~~
  - ~~**75** stars~~
  - ~~(if you want to *brew install jr*!)~~

> more | grep links

- Links
  - Issues https://github.com/ugol/jr/issues
  - Documentation https://jrnd.io/
  - Blog first part: https://dev.to/ugol/jr-quality-random-data-from-the-command-line-part-i-5e90
  - Blog second part: https://dev.to/ugol/jr-quality-random-data-from-the-command-line-part-ii-3nb3

> more | grep questions?

CONFLUENT