CHAPTER 07

GenRocket (2): data patterns and a hands-on build

Sequential, random, boundary, edge-case and null data, then a full lab generating 50 customer records.

GenRocket: data patterns for realistic and tricky test data

Beyond the building blocks of GenRocket (the overview, domains, attributes and generators, receivers and scenarios), the question is *which* values a generator should produce so that tests find bugs.

The bugs live at the edges: an order total of exactly 0.00 or 999999.99, a customer exactly 18 years old, an empty email, a phone number with a digit too few. Testers call these deliberate shapes **data patterns**. The patterns covered are sequential, random, boundary and edge cases, and null. Each gets a plain-words explanation with a ShopSphere example, then how GenRocket does it with documented generators, then a coverage table and a boundary-value lab. The next chapter uses several of these patterns in a full hands-on lab.

7.1 Words to know first

Positive test

A test that feeds the app data it should accept, to check it works. Example: a valid order of 49.99.

Negative test

A test that feeds the app data it should reject or handle gracefully, to check it fails safely. Example: an order total of -5.00.

Sequential data

Values that follow a fixed order, such as 1, 2, 3, 4. The usual choice for ids and numbered records.

Random data

Values picked unpredictably from a set or range. Good for realistic variety.

Seed

A starting number for the random picker. Same seed in, same "random" values out, every run. This is how random data becomes repeatable.

Boundary value

A value at, just below or just above the limit of what is allowed. The limit is the "boundary".

Edge case

A rare or unusual situation outside the typical pattern, such as a name longer than the column allows. GenRocket's own definition is "a rare or unexpected scenario outside the typical usage pattern".

Null

"No value at all" in a database column. Different from an empty string, which is a value that happens to contain zero characters.

7.2 Why test data needs patterns

A vending machine tested only with the exact price always passes; bugs appear with one coin too few, one too many, or none. Software is the same: defects hide at a limit, an empty field or a rare combination. So testing needs **typical data** (positive tests: looks real, proves the main flows) and **tricky data** (negative and edge tests: sits on the rules, proves the app rejects bad input safely). A production copy is poor at the second kind: it may hold no order of exactly 0.00 and no 17-year-old customer. With synthetic data you decide what exists.

GenRocket's CI/CD page says its web platform can design "any form of test data (positive, negative, patterns, permutations, boundary values, dynamic, load, etc.)" for a data model (Automating Data Generation using CI/CD Pipeline).

7.2.1 The ShopSphere patterns we will build

The examples use the fictional ShopSphere tables. The table lists the patterns wanted and where each is built.

SHOPSPHERE DATA PATTERNS AND THE SECTION THAT BUILDS EACH (EXAMPLE DATA)

Column	Pattern wanted	Example values	Built in
`customer_id`, `order_id`	Sequential, unique	1001, 1002, 1003	section 7.3
`first_name`, `last_name`	Random, realistic	Asha, Menon	section 7.4
`phone`	Sequential valid numbers; a few too short, too long or empty	(800) 600-1000; 98765; (empty)	section 7.3, section 7.7
`order status`	Weighted: most Delivered, some Pending	Delivered 70 percent	section 7.5
`total_amount`	Typical, plus boundaries	49.99; 0.00; 0.01; 999999.99	section 7.8
`email`	Mostly valid; some null; some empty	user1@example.com; NULL; (empty)	section 7.7
`date_of_birth`	Typical adults, plus exactly 17 and 18 years old on a reference date	1990-04-12; 2008-01-01	section 7.8, section 7.10
`first_name` length	Mostly short, some longer than the column allows	"Asha"; a 12-letter name	section 7.6

7.2.2 How GenRocket thinks about a pattern

In GenRocket a pattern is a *generator choice plus its parameters* on an attribute; there is no separate "pattern" switch. You pick a generator (see the chapter on domains, attributes and generators), set its parameters and read the Preview pane. Generators only create raw values; receivers format them. Three mechanisms cover this section: one generator with parameters (RangeGen, RandomGen, ListGen), several generators linked in one attribute, and attribute settings such as `% Null` that sit outside the generator. The help centre says GenRocket has over 765 generators, so only the relevant ones are covered.

7.2.3 What the vendor says patterns are for

In its 2022 overview video, GenRocket's CEO says masked copies of production data leave "data gaps" that enterprise customers report: no unique data, no negative data, not every permutation, too little

volume, nothing in a required condition. His answer is data designed to order: positive and negative data, *unique permutations* (every useful combination of values), and *stateful data* (rows in a particular condition, such as an order already shipped), always with referential integrity, shaped by rules and delivered in any volume. In the 2020 webinar GenRocket's CTO adds a design principle: most generators do not produce purely random values, because random data is hard to test with. They produce patterns, and linked generators combine them (section 6.18.12). These are the vendor's claims about its own product. Sequential, weighted and conditional data are the patterns; boundary, edge, null and permutation data are the tricky cases. The concepts chapter lists the same coverage in tool-neutral terms.

7.3 Sequential data

7.3.1 In plain words

Sequential data follows a fixed, predictable order: 1, 2, 3, 4, like numbered tickets at a bakery counter.

Testers want sequential data in three places:

- **Primary keys.** A primary key must be unique, so counting up is the simplest safe pattern. ShopSphere's `customer_id` values 1001, 1002, 1003 are sequential.
- **Numbered records.** Order numbers such as ORD-1001, ORD-1002, where you want to look at row 5 and know exactly what should be there.
- **Predictable walks.** A list of statuses walked in order (Pending, Shipped, Delivered) so a test can assume which row has which status.

Sequential data is easy to read when a test fails, but looks nothing like real life, so mix it with random data for other columns.

7.3.2 RangeGen: counting up (and down)

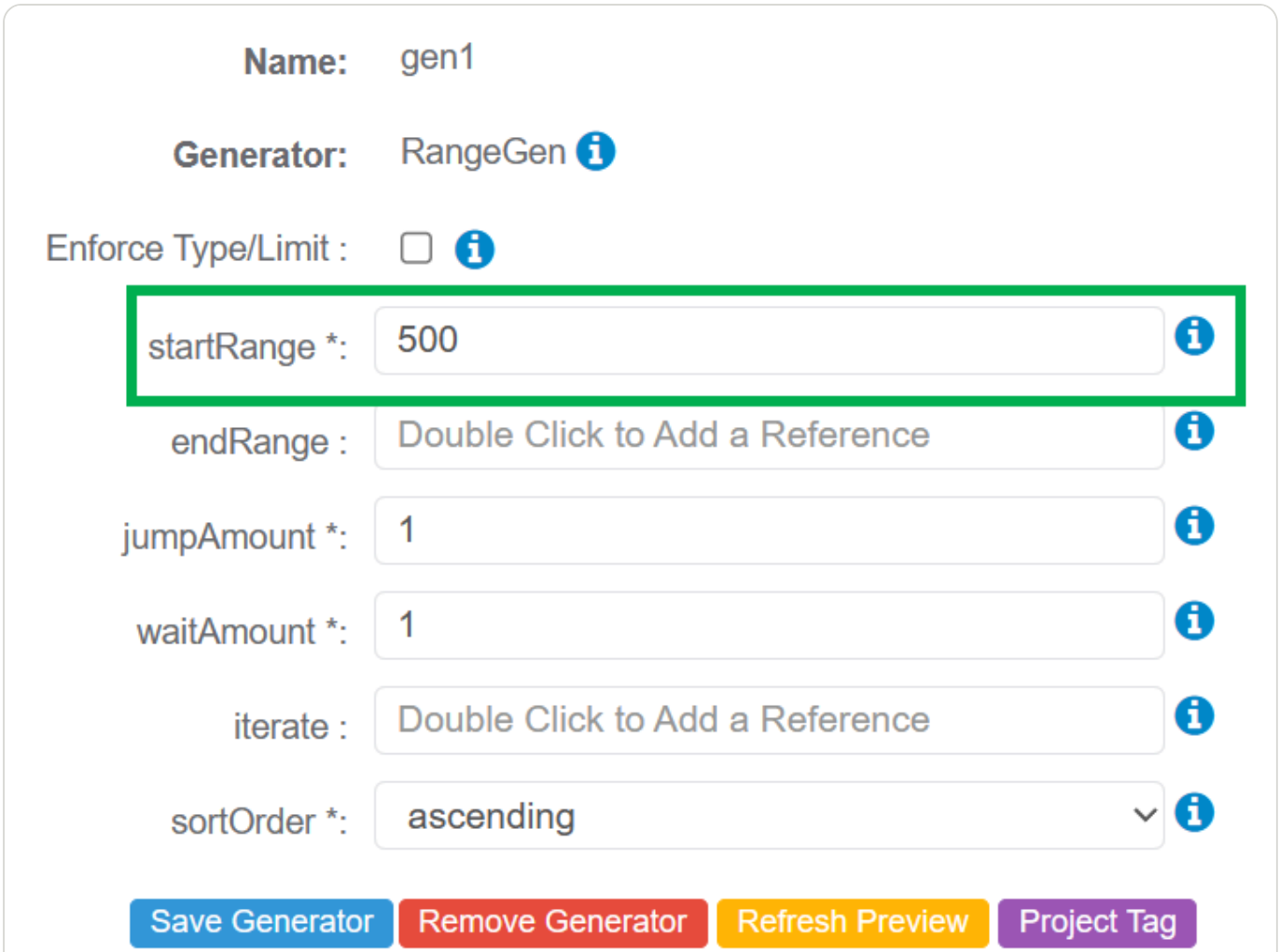
The generator for counting is **RangeGen**. The help page says it "produces an increasing value from startRange to endRange", or to `long.MAX_VALUE` when no end range is given, and that "once the endRange has been reached, the value is reset to the startRange" (RangeGen help page). The page also notes it can be set to start at a number other than 1, for example to continue a database's primary keys. The 2020 webinar demo shows the pattern live: the `id` counts up from 1,000, and changing the jump from 1 to 3 makes the sequence step by three (section 6.18.11).

RangeGen parameters as documented (asterisk means required)

Parameter	What it does	ShopSphere use
`startRange*`	The first value.	1001 for `customer_id`.
`endRange`	The largest value before the count restarts at `startRange`. If empty, it is `long.MAX_VALUE`.	Leave empty for a primary key so it never wraps.
`jumpAmount*`	How much to add each time.	5 to leave gaps between ids.
`waitAmount*`	How many rows to wait before adding `jumpAmount`.	3 to give each customer three rows (for example three order items).
`iterate`	A true/false that decides when to move to the next value. Can reference another generator that produces true/false.	Advance only when a linked generator says true.
`sortOrder*`	`ascending` or `descending`.	Descending for negative ranges.

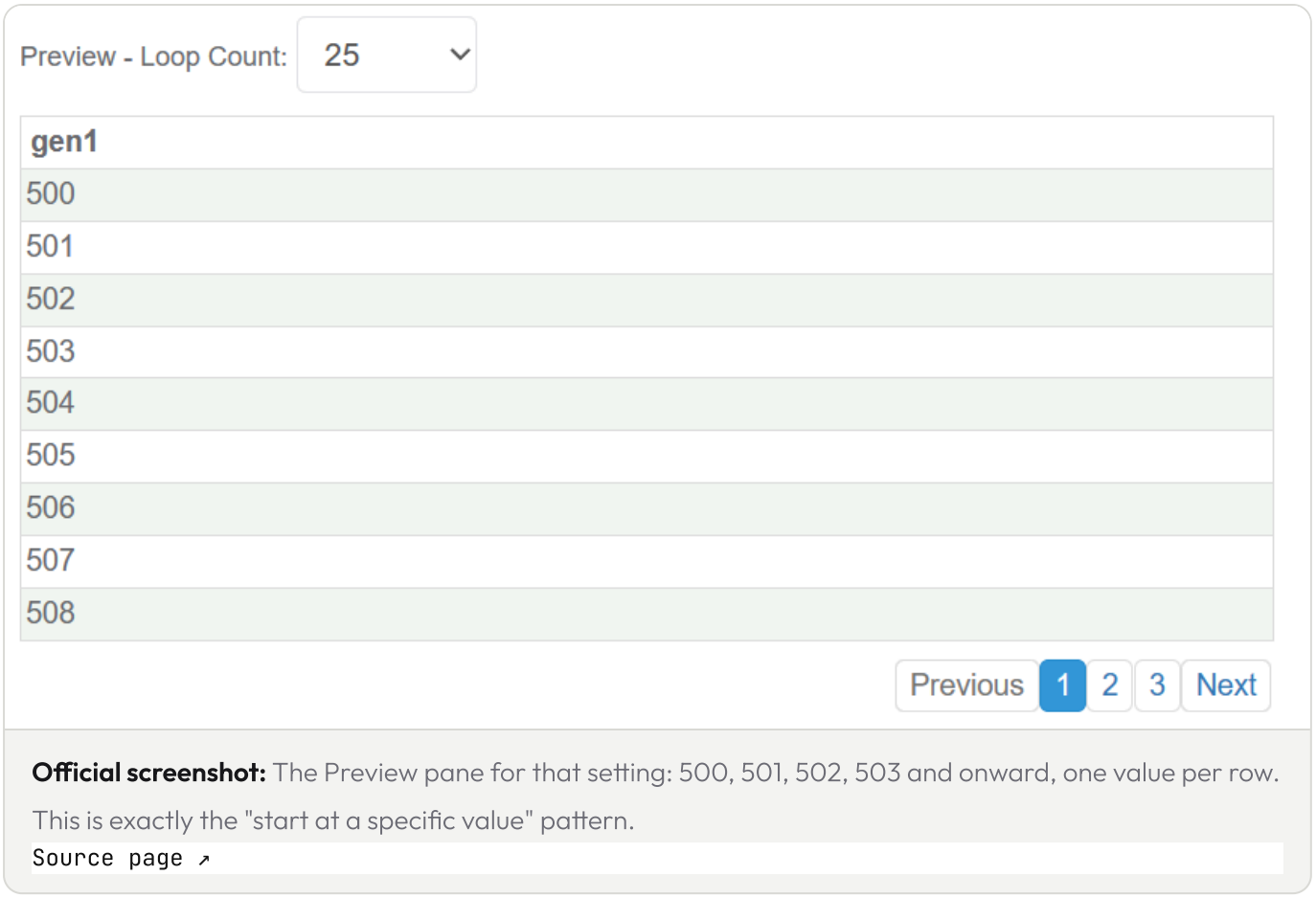
The help page shows five worked examples. Four of them follow, each with the official screenshots of the parameters and the preview.

Start at a chosen value. Setting `startRange` to 500 makes the first row 500, then 501, 502 and so on.



Official screenshot: The RangeGen parameter form with `startRange` set to 500, `jumpAmount` and `waitAmount` both 1, and `sortOrder` ascending.

`Source page ↗`



Official screenshot: The Preview pane for that setting: 500, 501, 502, 503 and onward, one value per row. This is exactly the "start at a specific value" pattern.

Source page ↗

Jump and wait. With `startRange` 0, `endRange` 100 and `jumpAmount` 5 the values are 0, 5, 10, 15 and so on. Raising `waitAmount` to 3 makes each value repeat three times (0, 0, 0, 5, 5, 5) before the jump, the pattern for one parent value repeated across several child rows.

Count down through negatives. Set `startRange` -1, `endRange` -100 and `sortOrder` descending.

negativeValues
-1
-2
-3
-4
-5
-6
-7
-8
-9
-10

Official screenshot: The preview for that setting: -1, -2, -3 down to -10. Negative numbers like these are the raw material for negative tests, such as a negative quantity.

`Source page ↗`

DIRECTION AND RANGE MUST AGREE

If you give a descending order a positive end range, GenRocket stops you. The help page names the error "startRange cannot be less than or equal to endRange" and gives two fixes: change `endRange` to a negative value, or change `sortOrder` to ascending. The official screenshot below is the red error box the form shows.

Generator Parameter Errors

- The startRange , -1, cannot be less than or equal to endRange, 100 for RangeGen in Attribute 'negativeValues'.

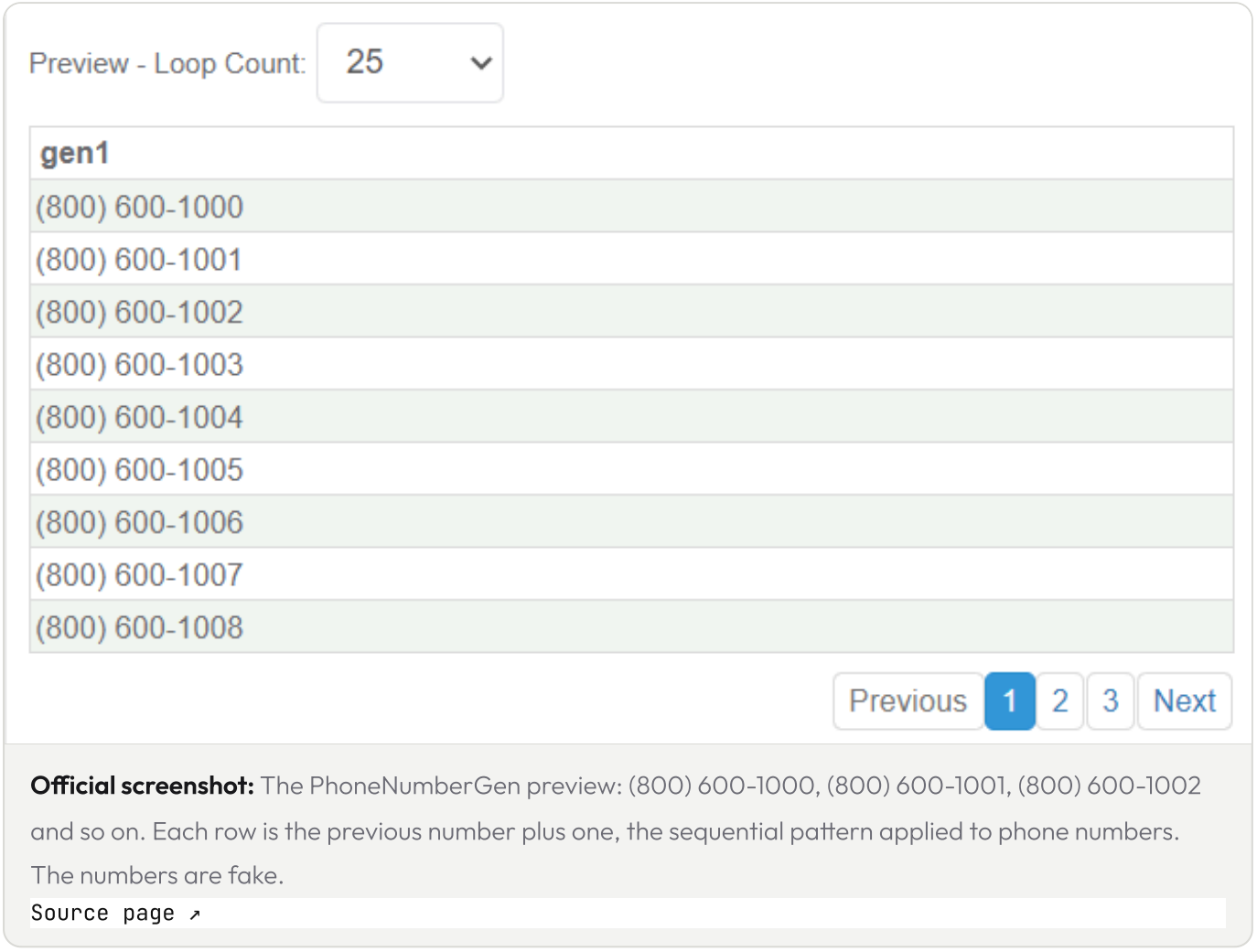
Official screenshot: The "Generator Parameter Errors" message shown when the start and end values contradict the sort order. It names the attribute and both values, so you can see which one to change.

`Source page ↗`

7.3.3 Other ways to get sequential values

RangeGen is the main tool, but the documentation shows several more sequential patterns.

- **The domain id.** When a domain is created or imported it automatically gets an `id` attribute that "generates a unique ID for each test data record" (Attributes Overview). For many tests that is the primary key and you need do nothing.
- **RangeConstantGen.** The string-generators list describes it as an increasing value "similar to the RangeGen" with an optional prefix and suffix. With range 1 to 5 and prefix `test` the output is test1 to test5. The StringRemoveGen page confirms the parameter names `startRange` and `prefix` for it. For ShopSphere, a prefix of `ORD-` and start 1001 gives ORD-1001, ORD-1002 and so on.
- **ListGen in order.** ListGen "generates items in a provided list in sequential or random order". Its `random` parameter defaults to False, so the list is walked in order. Its example sets `waitAmount` to 5 so each of Pending, Shipped and Delivered repeats for five rows before moving on (ListGen).
- **PhoneNumberGen.** The page says it "generates phone numbers in a sequence (100-1000, 100-1001, ..., 999-9999)", with two optional parameters: `areaCode` (the three-digit area code placed in front) and `prefix` (the starting prefix, from which the number increases). It points to RandomPhoneNumberGen for random numbers.



Official screenshot: The PhoneNumberGen preview: (800) 600-1000, (800) 600-1001, (800) 600-1002 and so on. Each row is the previous number plus one, the sequential pattern applied to phone numbers. The numbers are fake.

Source page ↗

7.3.4 Continuing after existing data

Sometimes the target database already has rows, and new primary keys must start after the largest existing one. The help centre shows a linked-generator recipe (primary key values after the max value):

1. **Add QueryOnceGen to the id attribute.**
   It runs one query before generating. Set `resourceName` to the JDBC resource, `query` to `select max(id) + 1 from <tableName>` and give `sampleValue` (the page uses 5001) so the Preview pane has something to show.

   Expected result: gen1 holds "the next free id".

2. **Add RangeGen after it.**
   For `startRange`, double-click the field and choose the Attribute Generator tab, then pick `gen1` (it is written `#{self.gen1}`).

   Expected result: New ids begin after the largest existing id, so inserts will not collide with existing rows.

RangeGen can reference QueryOnceGen only because QueryOnceGen sits to its left, a rule covered in section 7.6.

> **GOTCHA: A SEQUENCE THAT WRAPS IS NOT UNIQUE**
>
> RangeGen goes back to `startRange` when it reaches `endRange`. If you set a small `endRange` on a primary key and generate more rows than the range holds, you create duplicate keys. Leave `endRange` empty for ids, or make sure the loop count is no larger than the range.

7.4 Random data and repeatable randomness

7.4.1 In plain words

Random data is data picked without a visible order. Random data is what makes a generated customer list look real: different names, ages and cities.

Random data has one practical problem. If a test fails today with a random customer, and tomorrow's run creates different customers, you cannot reproduce the failure. The fix is the **seed**: a starting number for the random picker. The same seed gives the same "random" sequence every time. Random but repeatable is the best of both.

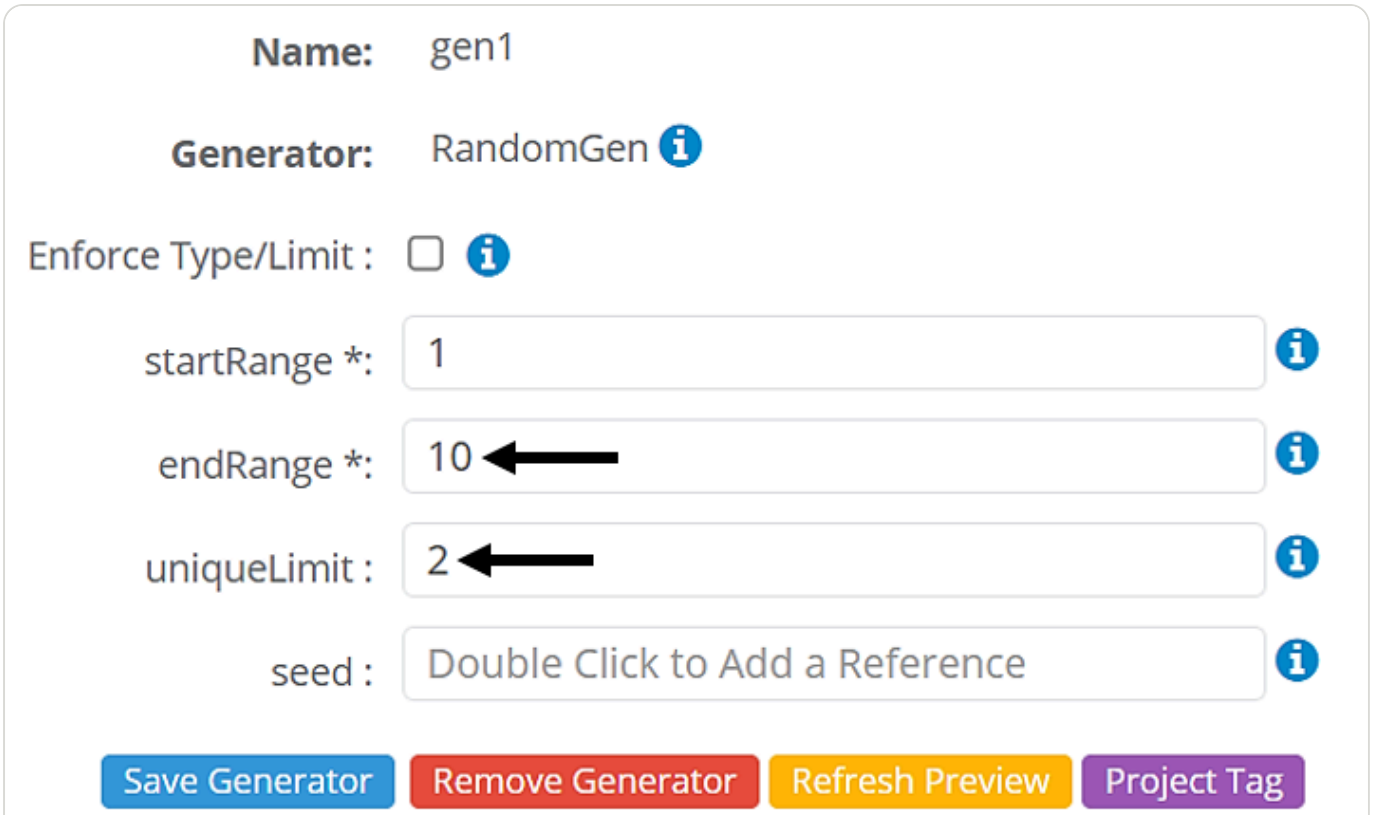
7.4.2 RandomGen: a random number in a range

RandomGen "randomly generates a number within a range"; the page's own example is random ages between 18 and 65 (RandomGen).

RandomGen parameters as documented

Parameter	Meaning
`startRange*`	The smallest value that can be produced.
`endRange*`	The largest value that can be produced.
`uniqueLimit`	How many times a given value may be produced. Empty means no limit. 1 means each value appears at most once; 10 means each value can appear ten times. It does not apply to dynamic start and end ranges that change per row.
`seed`	Makes the same random data come out every run.

The page's second example limits repeats: numbers 1 to 10, `uniqueLimit` 2, and a domain loop count of 15 so the possible values are not used up. Each number appears at most twice.



Official screenshot: RandomGen set to `startRange` 1, `endRange` 10 and `uniqueLimit` 2, so no number may appear more than twice.

`Source page ↗`

id	number
1	7
2	2
3	10
4	4
5	8
6	2
7	1
8	4
9	9
10	10
11	9
12	7
13	1
14	3
15	8

Official screenshot: The delimited output for that setting: 15 rows, numbers between 1 and 10, none more than twice. This is how you generate "random but not too repetitive" values.

`Source page ↗`

THE ERROR YOU GET WHEN UNIQUENESS RUNS OUT

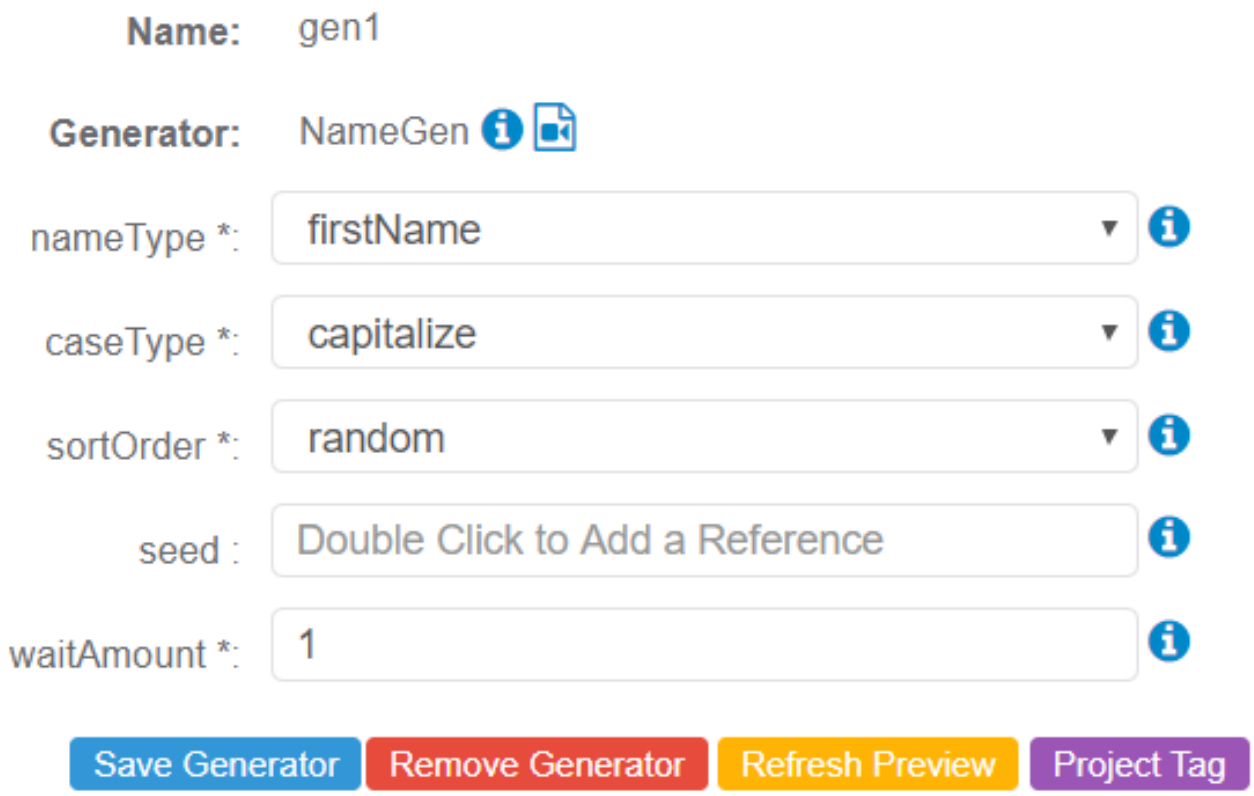
With `uniqueLimit` 1, range 1 to 20 and a loop count of 20, the page reports *"Exhausted attempts to find the next valid random value from range of 1 to 20"*. The fixes it lists: raise the loop count, widen `endRange` or raise `uniqueLimit`. The rule of thumb is that rows asked for must be no more than (range size times `uniqueLimit`). The same Java memory exhaustion can appear for very large unique sets: the troubleshooting page for "GC overhead limit exceeded" lists "generator parameters that limit the amount of data, such as a unique limit" as a cause (that error page).

7.4.3 Random names, lists and strings

- **NameGen.** Generates first, last, male or female names (`nameType`), cased as `capitalize` , `upperCase` or `lowerCase` (`caseType`), in `random` , `ascending` or `descending` order

(`sortOrder`), with an optional `seed` and a `waitAmount`. Its pool (1,000 male, 3,000 female and 10,000 last names) gives 40,000,000 combinations.

- **ListGen.** Set `random` to True to pick from a fixed list in random order (the page example picks one of five colours). Add `unique` True to use each list value only once; then the loop count must not exceed the number of values, or GenRocket throws an exception. ListGen supports up to 15 list items; for bigger lists the page recommends ListCSVV2Gen.
- **RandomStringGen.** Random text of a given `length` and `stringType` (letters, numbers, lettersAndNumbers, specialCharacters, ASCIICharacters or allCharacters), with `caseType`, `seed`, `unique`, `include` and `exclude`. The help page lists "test boundary conditions or invalid characters" and "filling a notes field with random strings to test character limits" among its uses. Its promo-code example links ConstantGen (PROMO), RandomStringGen (6 unique digits) and ConcatGen.



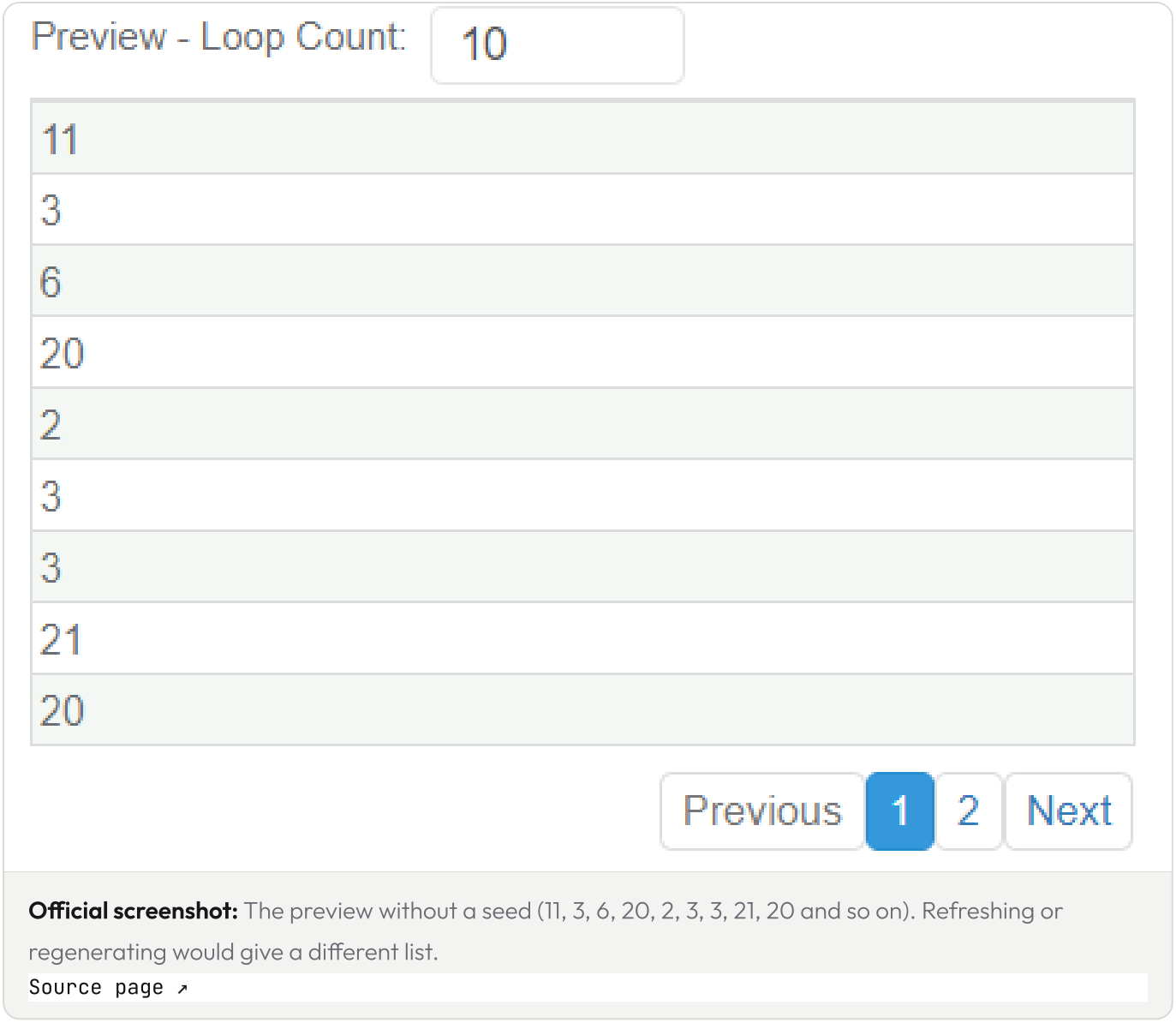
Official screenshot: NameGen with `nameType` firstName, `caseType` capitalize, `sortOrder` random, an empty `seed` and `waitAmount` 1. This is the setting used for first names in the 50-records lab.
`Source page ↗`

7.4.4 Seeding: random, but the same every time

The help page "What is Seeding?" defines seeding as generating "the same set of random data for the same value of a seed". It says a seed can be entered in several places in GenRocket, including the domain's

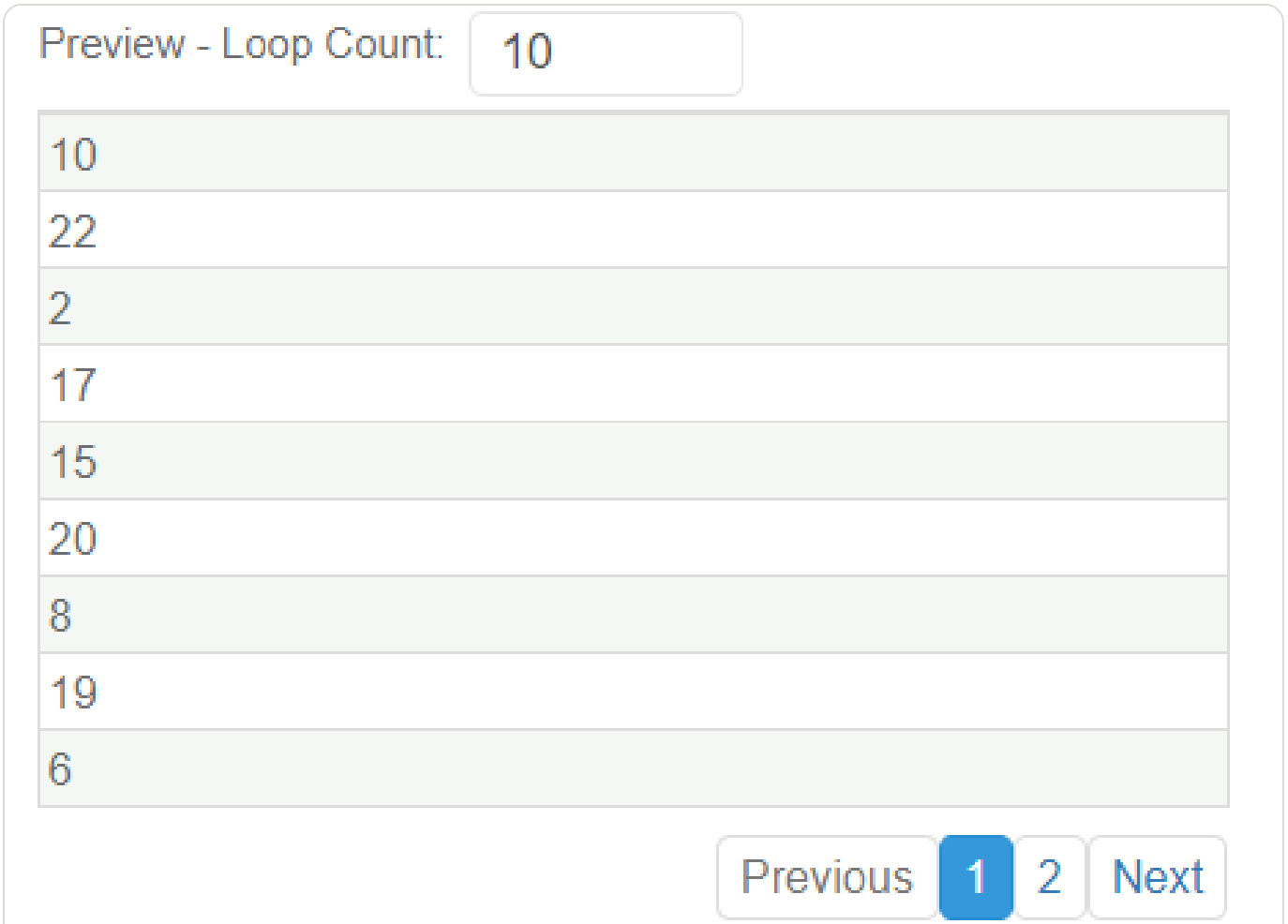
Global Seed variable and the generator's own `seed` parameter (What is Seeding?). The page's example uses two attributes, `number1` and `number2`, both RandomGen from 1 to 25:

- No seed on `number1`: a different set of numbers every time the data is generated.
- Seed 1 on `number1`: one fixed set, the same every run.
- Seed 1 on `number2` too, same configuration: `number2` gets the same set as `number1`, so use different seeds for columns that should look independent.
- Seed 2 on `number2`: a different fixed set.



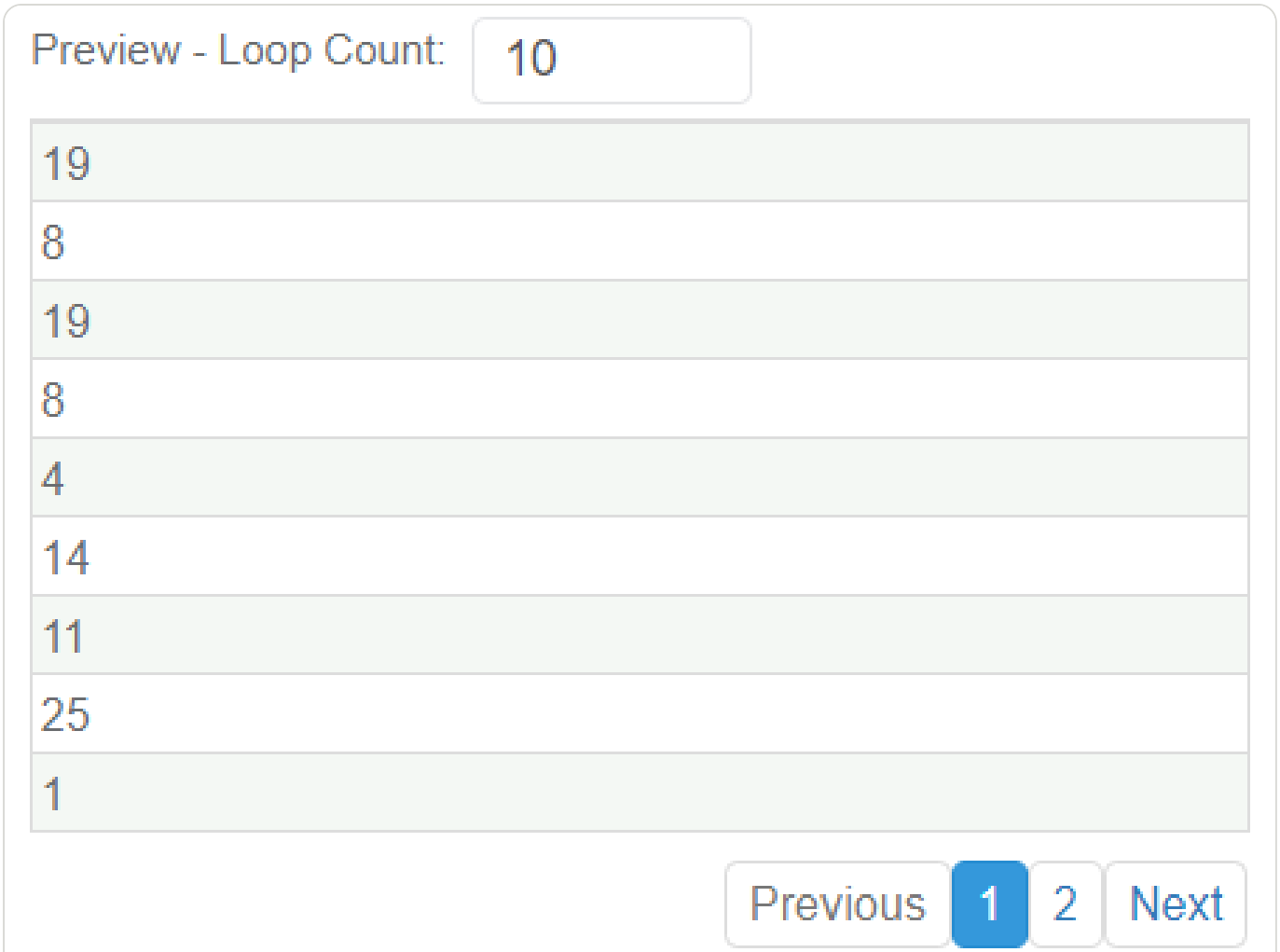
Official screenshot: The preview without a seed (11, 3, 6, 20, 2, 3, 3, 21, 20 and so on). Refreshing or regenerating would give a different list.

`Source page ↗`



Official screenshot: The preview with seed 1: 10, 22, 2, 17, 15, 20, 8, 19, 6. With this seed these are the values you would get on every run.

`Source page ↗`



Official screenshot: The preview with seed 2: 19, 8, 19, 8, 4, 14, 11, 25, 1. It differs from the seed 1 list, and it will be the same on every run of seed 2.

`Source page ↗`

The 2020 demo shows the problem live. With no seed, the first name, last name and middle initial change every time the preview refreshes. The CTO explains that a global seed, which can be assigned to many generators, makes the same random data appear every time for the same seed value. That matches the docs' domain Global Seed and generator `seed`.

WHY SEEDING MATTERS FOR A TDM ENGINEER

Without a seed, "random" test data differs each run, so a failing test is a ghost. With a fixed seed, a regression suite sees identical data every night, and a failure can be replayed on a developer's laptop. Seed attributes you want repeatable. Leave the seed empty on attributes where you want fresh variety each run, such as a load test.

7.5 Lists, weights and percentages

7.5.1 In plain words

Real data is lumpy. In a real shop, most orders are Delivered, a few are Pending and a handful are Cancelled. If your generator gives each status the same share, your test data is unrealistic, and reports built on it (percentage cancelled, say) look wrong. A **weighted** pattern says "give value A to 70 percent of the rows and value B to 30 percent".

There are two related ideas. A **list** is the set of allowed values (Pending, Shipped, Delivered). A **weight** says how often each list value should appear. Lists come first, weights shape them.

A SHOPSPHERE ORDER-STATUS MIX

Status	Share of orders	Why that share
Delivered	70 percent	Most historical orders are finished.
Shipped	15 percent	In transit.
Pending	10 percent	Just placed.
Cancelled	5 percent	Rare, but the app must handle it.

7.5.2 ListGen: the allowed values

ListGen "generates items in a provided list in sequential or random order" and is "a great choice when generating data from a fixed set of values (e.g., lookup table or categorical data)". Its documented parameters are `list` (one item per line), `random` (False by default), `unique`, `waitAmount`, `iterate`, `seed`, `resetIndex` and `setLoop`. It supports up to 15 items.

One parameter matters for the boundary lab at the end of this section. `setLoop` "gives users the ability to OVERRIDE the loop count at the Scenario Level and ONLY produce the number of rows defined in this list". If the first generator of an attribute is a ListGen with `setLoop` True, the scenario's loop count is locked to the list length and shows as "N" with no edit pencil. The help page "Why can't I update the Loop Count Value from the Scenario?" explains that this lock is the reason (link).

7.5.3 WeightGen: a two-way split

WeightGen "defines a percentage between 1 and 100 to determine how often values generated from variable1 will be selected over values generated from variable2". The parameters are `percentage*`,

`variable1*`, `variable2*` and an optional `seed`. Variables can be constants or references to other generators or attributes (WeightGen).

The page's example: `percentage` 70, `variable1` Pass, `variable2` Fail gave 7 Pass and 3 Fail in ten rows. WeightGen splits between exactly two things; for more use MultiWeightGen.

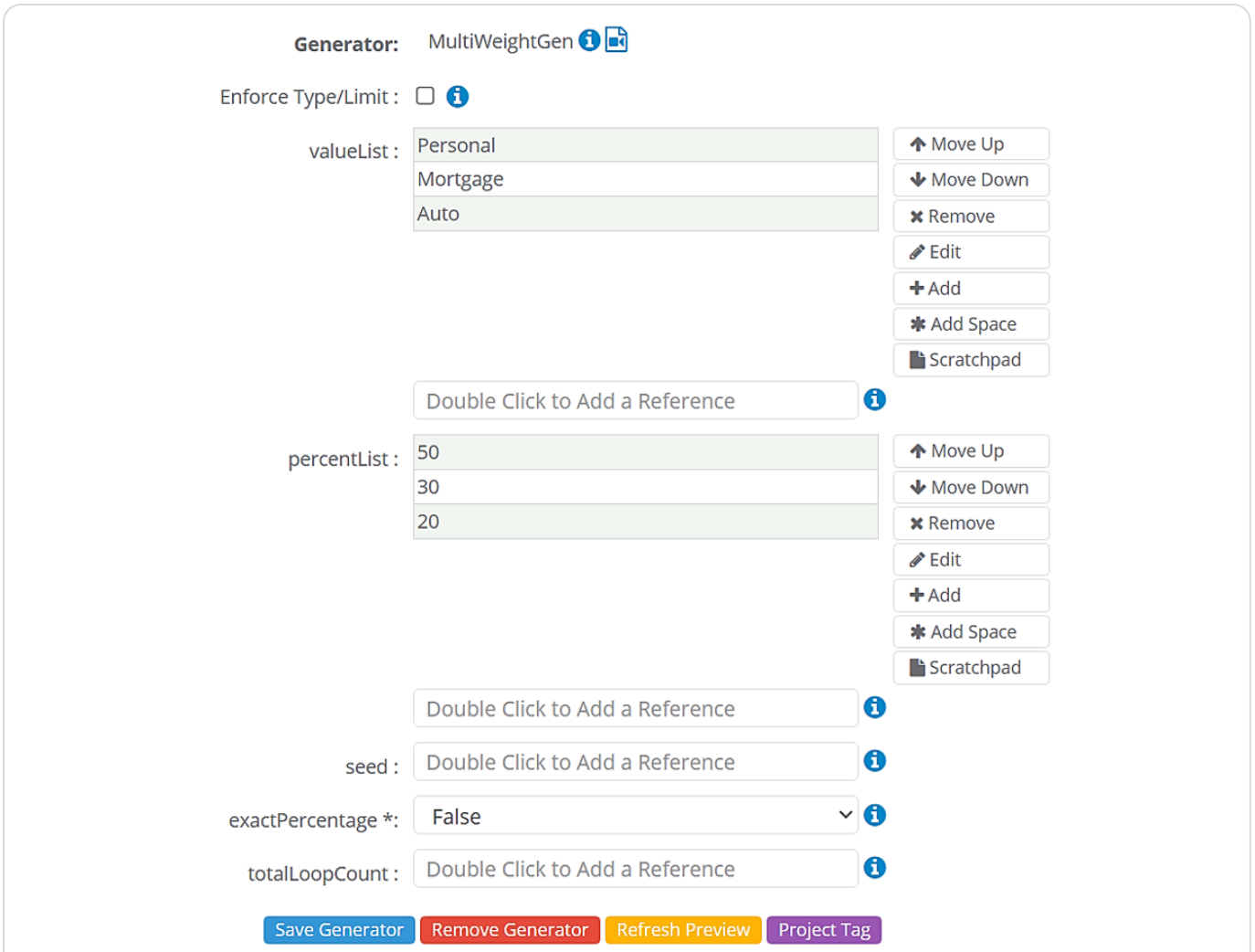
7.5.4 MultiWeightGen: any number of weighted values

MultiWeightGen "allows multiple values to be weighted to determine how often each value is generated". The page's own example is rental vehicle types: economy 50 percent, midsize 35 percent, luxury 15 percent (MultiWeightGen).

MultiWeightGen parameters as documented

Parameter	Meaning
`valueList`	The values. Press Enter after each. Each value is paired with the percentage in the same position.
`percentList`	The percentages (whole numbers) or references to values holding them. The page says to make them add up to 100 and notes the generator tolerates more than 100 in rare cases.
`seed`	Same random data every run.
`exactPercentage*`	When True, the distribution matches the percentages exactly. When False it is decided as rows are generated and may be off, with larger row counts converging on the percentages.

With 100 records and `exactPercentage` True, the page says you get exactly 50 economy, 35 midsize and 15 luxury (one screenshot on that page shows `exactPercentage` False, so set it deliberately).



Official screenshot: The loan-type example: `valueList` Personal, Mortgage, Auto with `percentList` 50, 30, 20.

`Source page ↗`

id	loanType	interestRate
1	Personal	5%
2	Mortgage	4%
3	Mortgage	5%
4	Personal	5%
5	Mortgage	4%
6	Personal	15%
7	Auto	6%
8	Personal	10%
9	Auto	8%
10	Personal	5%

Official screenshot: The output of the page's second example: a `loanType` column chosen by weight, and an `interestRate` column that depends on it (Personal rows show 5%, 10% or 15%; Mortgage rows 3% to 5%; Auto rows 6% to 8%). One column driving another is the linked pattern in the next section.
`Source page ↗`

Note that MultiWeightGen "works in the totality of all rows and does not consider parent/child relations"; the overview names MultiWeightChildGen for that case.

For ShopSphere's status mix (Delivered 70, Shipped 15, Pending 10, Cancelled 5) use the same recipe with `exactPercentage` True and a `seed`; with 1,000 orders expect exactly 700, 150, 100 and 50.

How to give each *parent* a different number of children by weight is covered by the help page "How to generate Children based on Weights (Percentages)", and it is built in the parent and child lab.

7.6 Linked and conditional generators

7.6.1 In plain words

Sometimes a value depends on another value. A voting-age check depends on age. A "refund reason" only makes sense if the order is cancelled. In a spreadsheet you would write an IF formula. GenRocket has the

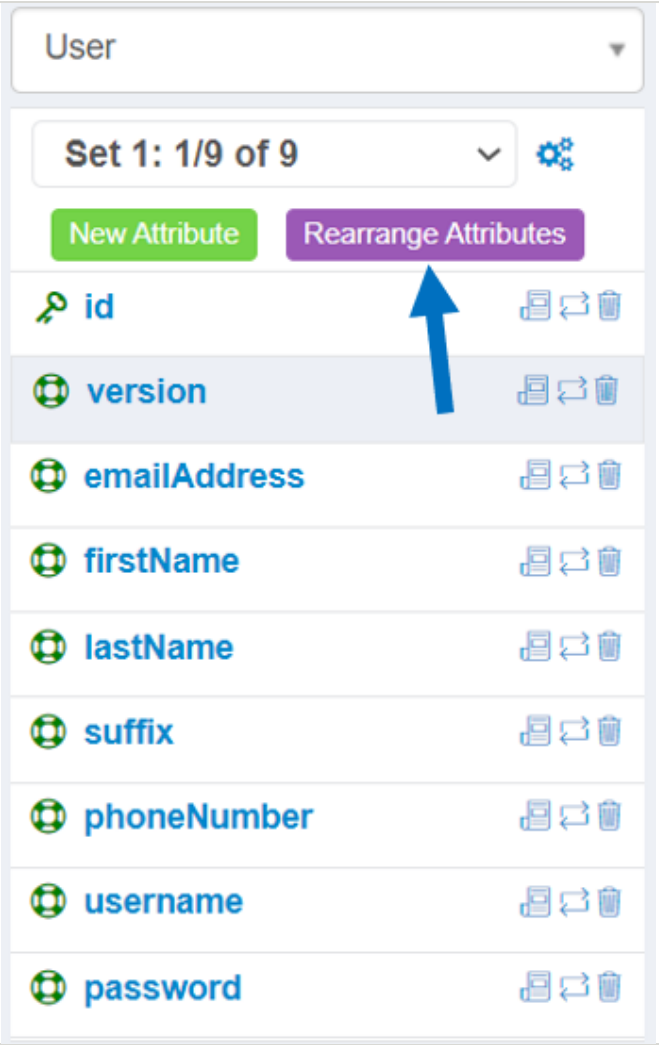
same idea in two parts: **linked generators** (several generators chained in one attribute) and **conditional logic generators** (generators that choose one result by testing a condition).

7.6.2 Linked generators: the left-to-right rule

The "Linked Generators Overview" page gives the rules that every conditional recipe relies on (Linked Generators Overview):

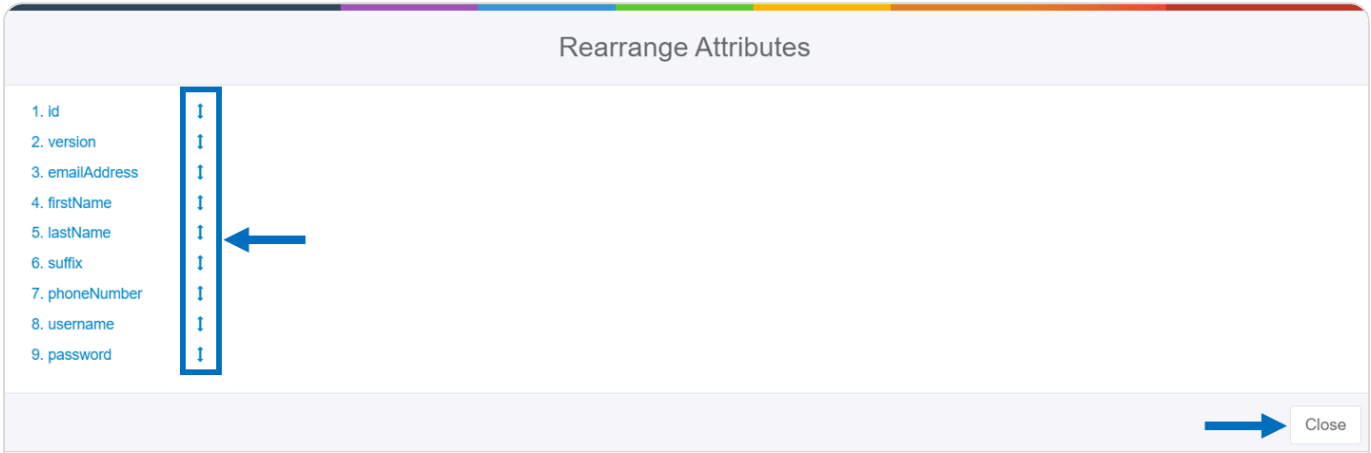
- Linked generators run **left to right** (top to bottom in the table): the first generates first, the second next, and so on.
- A linked generator may only reference **sibling generators to its left**. Referencing one to the right "will yield null values" because it has not generated yet.
- The **rightmost** generator holds the value returned for the attribute. The earlier ones are workings that do not appear in the output.
- Each one gets a default name (`gen1` , `gen2` and so on) and you can add an alias. In a parameter, a reference to a sibling looks like `#{self.gen1}` , as in the ConcatGen design-pattern page.

The same ordering applies between attributes: "Attributes can only reference other Attributes that are above them in the Attribute pane of a Domain." An attribute that references one below it "will return an empty value". The page "When can an Attribute not reference another Attribute?" shows the fix: click Rearrange Attributes and drag the attribute (link).



Official screenshot: The attribute list of a User domain (id, version, emailAddress, firstName, lastName and so on) with an arrow pointing at the **Rearrange Attributes** button.

`Source page ↗`



Official screenshot: The Rearrange Attributes dialog: a numbered list with a drag handle beside each attribute and a Close button, which saves the new order.

`Source page ↗`

7.6.3 The conditional logic generators

The page "Conditional Logic Generators" groups them into calculation, condition-based and percentage-based generators (link). The table below keeps the descriptions short; check the generator's own page for parameters.

Conditional logic generators and a ShopSphere use for each

Generator	What the docs say it does	ShopSphere use
EvalCaseGen	Evaluates one equation using up to 5 variables; returns `caseTrue` or `caseFalse`. Recommended.	If age >= 18 then "Yes" else "No".
EvalCaseActionGen	Several if-then-else cases; returns the first true. Recommended.	Loyalty tier by points.
SwitchGen	Like a Java switch: returns the value of the matched case. Recommended.	Shipping fee by country code.
EdgeCaseGen	Generates edge-case data for negative testing.	A negative age every few rows.
WeightGen / MultiWeightGen	Percentage-based choice.	Order status mix.

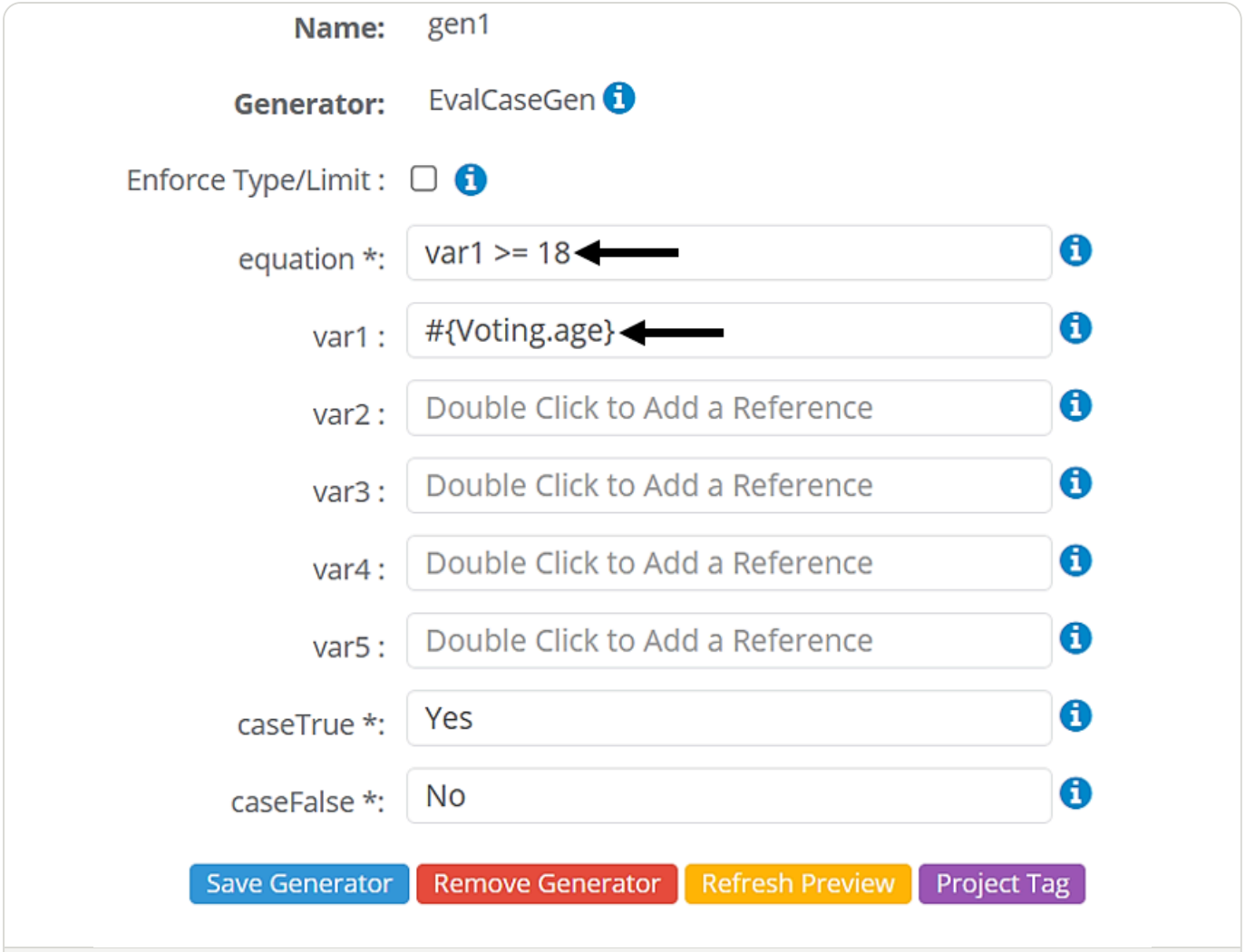
The overview adds that G-Rules can hold the same logic with conditions and actions, which suits complex multi-condition data. In the 2020 Q&A the CTO points to the same feature, "test data rules", for really complex if/then/else conditions, and to the linked case, switch and percentage generators for simpler ones.

7.6.4 EvalCaseGen: if age is 18 or more

EvalCaseGen takes an `equation` that uses `var1` to `var5`, plus `caseTrue` and `caseFalse`. Allowed operators include `&&`, `||`, `!`, `⩾`, `⩽`, `==`, `≠`, `<`, `>`, `%` and `/`. Two cautions from the page: put string comparisons in double quotes (for example `var1 == "Male"`), and for numbers of type long append a lower-case L (`var1l`) (EvalCaseGen).

Its voting example is a perfect small boundary test. A domain `Voting` has an `age` attribute (RandomGen from 10 to 65) and an `allowedToVote` attribute using EvalCaseGen with `equation` `var1 ⩾ 18`, `var1`

referencing the age attribute, `caseTrue` Yes and `caseFalse` No. Notice that the attribute order matters: `age` sits above `allowedToVote`.



Official screenshot: The EvalCaseGen form: `equation` `var1 ⩾ 18`, `var1` referencing the Voting age attribute, `caseTrue` Yes and `caseFalse` No.

Source page ↗

id	age	allowedToVote
1	29	Yes
2	22	Yes
3	14	No
4	46	Yes
5	45	Yes
6	20	Yes
7	11	No
8	50	Yes
9	24	Yes
10	47	Yes

Official screenshot: The output: ages such as 29, 22, 46, 45 show Yes, while 14 and 11 show No. Ages 18 and above are accepted, below are rejected.

Source page ↗

7.6.5 EdgeCaseGen: sneak in the tricky value

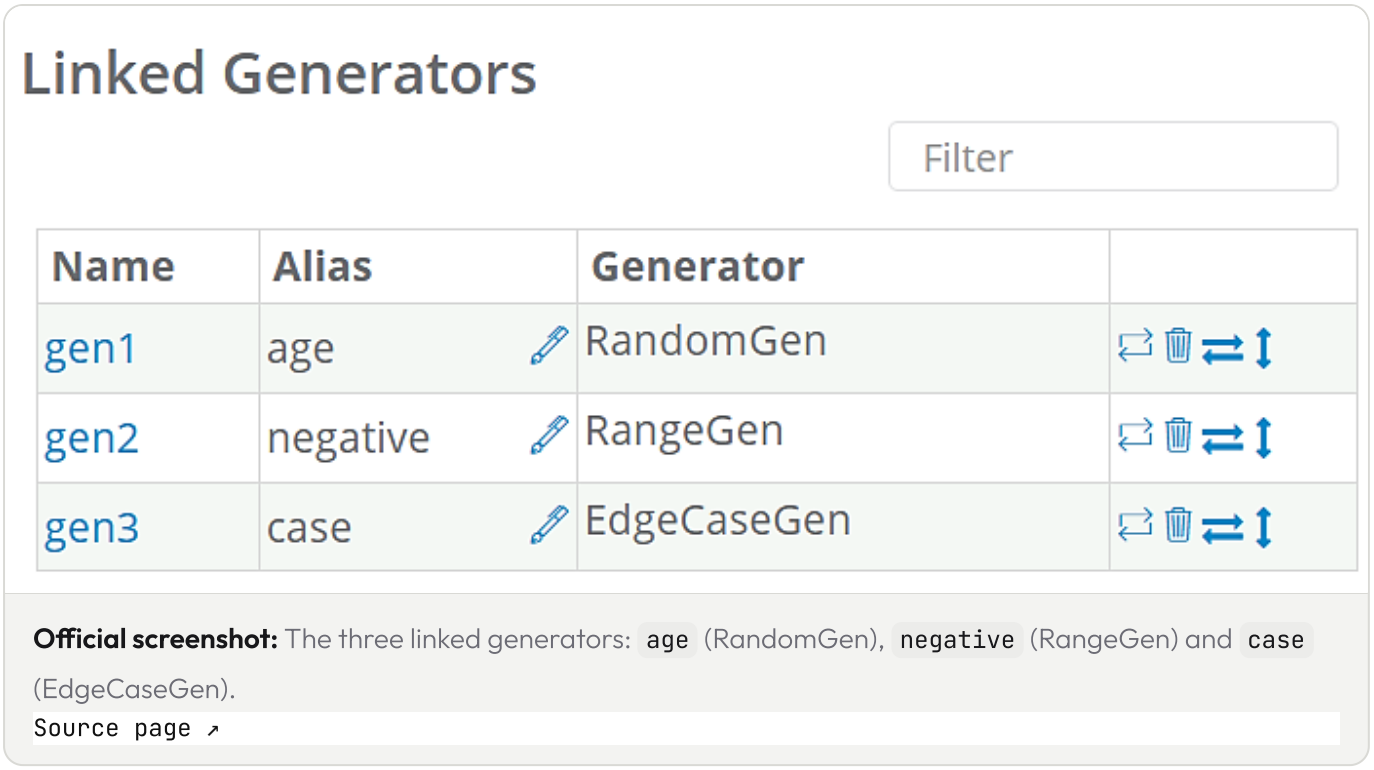
EdgeCaseGen mixes a normal value with an edge-case value. It "generates edge case data useful for negative testing and testing edge cases". The page lists example edge cases: rounding issues in currency conversion, mismatched time zones, special characters in passwords, names longer than the character limit, names with invalid characters, uncommon file formats and validating a user's age (EdgeCaseGen).

EdgeCaseGen parameters as documented

Parameter	Meaning
`source*`	The value (or reference) returned normally, when the edge-case condition is false.
`edgeCase*`	The value (or reference) returned when the edge case applies.
`condition`	A reference to a generator that returns true or false. True returns the edge case, false returns the source.
`cycle`	How many cycles the source value is returned before the edge-case value is returned.
`percentage`	The percentage of time the edge-case value is returned.
`seed`	Seed for the internal random picker used with `percentage`.

So there are three ways to decide when the edge case appears: a true/false condition, a fixed cycle, or a percentage.

Example 1 from the page: a negative age every few rows. Three linked generators: RandomGen `age` (1 to 120), RangeGen `negative` (-120 up to -1) and EdgeCaseGen `case` with `source` the age, `edgeCase` the negative value and `cycle` 5.



Official screenshot: The three linked generators: `age` (RandomGen), `negative` (RangeGen) and `case` (EdgeCaseGen).

`Source page ↗`

age	negative	case
88	-120	88
76	-119	76
112	-118	112
51	-117	51
59	-116	59
104	-115	-115
117	-114	117
116	-113	116
37	-112	37

Official screenshot: The preview with three columns. The `case` column repeats the `age` value for five rows (88, 76, 112, 51, 59) and then shows the negative value -115 on the sixth row, so the edge value arrives after five source values. Read the screenshot rather than assuming "every fifth row".

`Source page ↗`

Example 2: a first name longer than 10 characters, 25 percent of the time. Two ListGens hold short and long names, and EdgeCaseGen has `percentage` 25.

id	firstName
1	Tom
2	Mary
3	Sherry
4	Fred
5	Maximiliano
6	Troy
7	Cheryl
8	Jim
9	Peleloperose
10	Crystal
11	Tom
12	Anavictoria
13	Oliviagrace
14	Fred
15	Michael
16	Troy
17	Cheryl
18	Marcoantonio
19	John
20	Crystal

Official screenshot: The output: twenty first names. Five of them are long (for example Maximiliano, Peleloperose, Anavictoria, Oliviagrace, Marcoantonio), which is the 25 percent edge share. A name field that holds only 10 characters should reject or truncate those rows.

Source page ↗

THE DOCUMENTATION DISAGREES WITH ITSELF HERE

The `condition` parameter is documented as "True returns the edge case". In the page's third example (RandomGen source, RandomStringGen edge case, RandomTrueFalseGen condition) the screenshot shows the opposite: rows with True carry the source value, rows with False the edge value. Do not guess: build it, read the Preview pane, and flip the condition if the result is reversed.

G-CASES AND PERMUTATIONS

A G-Case "modifies the components within a Scenario", so one scenario can produce typical data in one case and edge-heavy data in another, with categories such as Beta, Unit, Load and Negative (G-Cases Overview). PermutationGen "can generate all possible combinations of data for a given set of values" from a `list` parameter; for example, drop-downs with 3, 4 and 2 choices give 24 combinations. Use boundary values for single fields and permutations for combinations of fields.

7.7 Null and blank values

7.7.1 In plain words

Imagine a paper form with a "middle name" box. There are three different situations. The person wrote a name (a value). The person wrote nothing and left the box empty (blank). Or the box was crossed out because the question did not apply (not applicable). Databases make the same distinction.

FOUR THINGS AN EMAIL COLUMN CAN HOLD

What is stored	Name	Typical bug it exposes
`asha.menon@example.com`	A value	None, this is the normal case.
(nothing, the cell is "unknown")	NULL	Code that calls a method on the email crashes with a null error.
`` (zero characters)	Empty string	Validation lets "" through because it is not null.
`  ` (only spaces)	Blank text	Trim logic missing, so the app thinks a value exists.

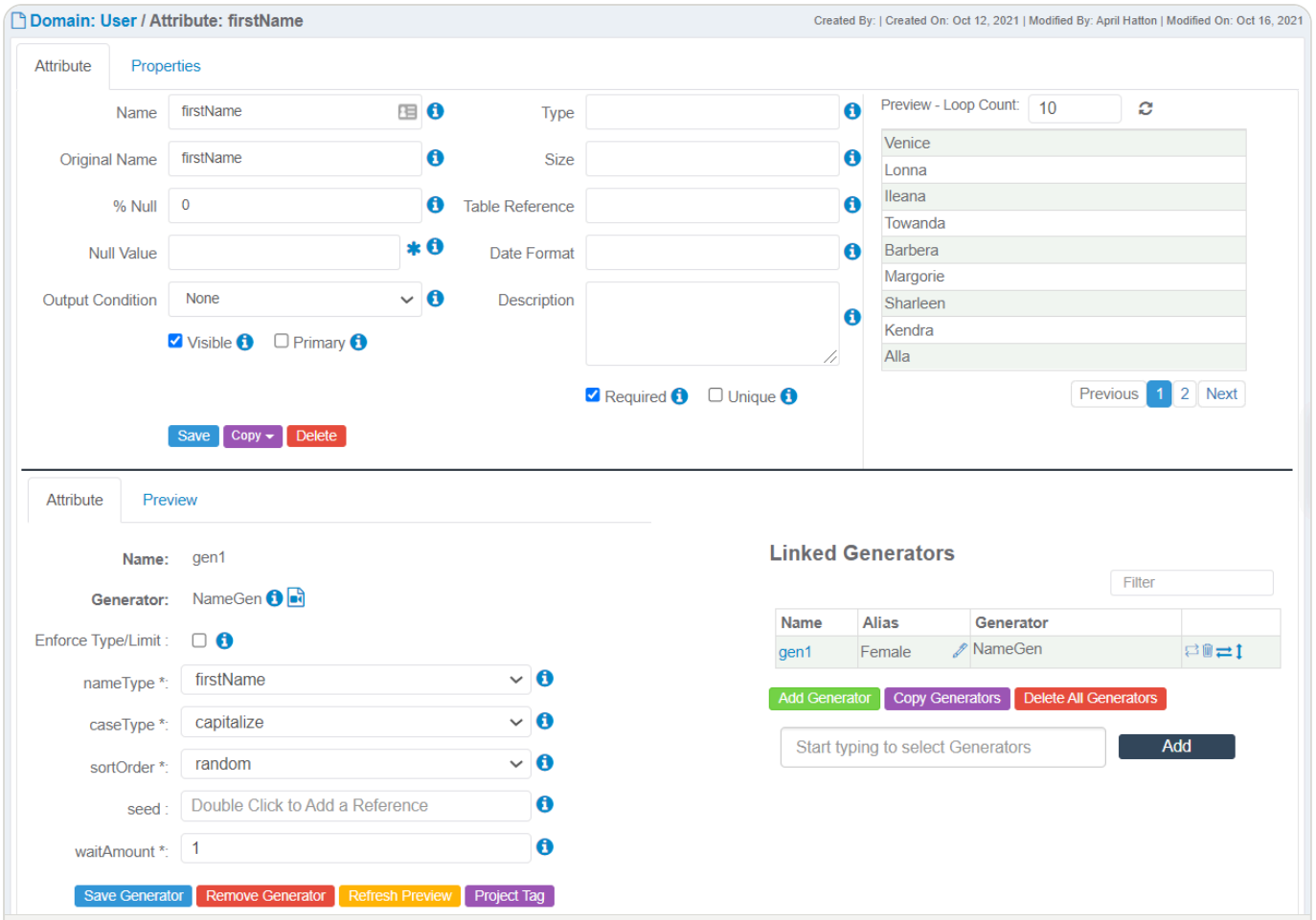
Applications treat these four differently, so test data needs all of them. GenRocket gives you three places to produce null and blank data.

7.7.2 Place 1: the attribute's own null settings

Every attribute has parameters on the Attribute Dashboard, above the generator. Four of them control nulls and visibility (Attribute Dashboard):

Attribute parameters that affect nulls and output, as documented

Parameter	What the docs say	Use
`% Null`	"Defines the percentage of time the null value is used for negative testing."	Make about 10 percent of emails null with one number.
`Null Value`	"Define the null value or click * for an empty string."	Choose what a null looks like. The asterisk means an empty string.
`Output Condition`	Whether to exclude the attribute from a receiver's output when its value is null, blank, or both.	Drop empty keys from JSON.
`Visible`	Visible to receivers when true, invisible when false.	Hide helper attributes from the output.



Official screenshot: The Attribute Dashboard for a `firstName` attribute. The top-left block holds the fields from the table: **% Null** (0 here), **Null Value** (blank, with a star button for "empty string"), **Output Condition** (None) and the **Visible** and **Primary** checkboxes. The generator form (NameGen) is underneath and a Preview pane sits on the right.

`Source page ↗`

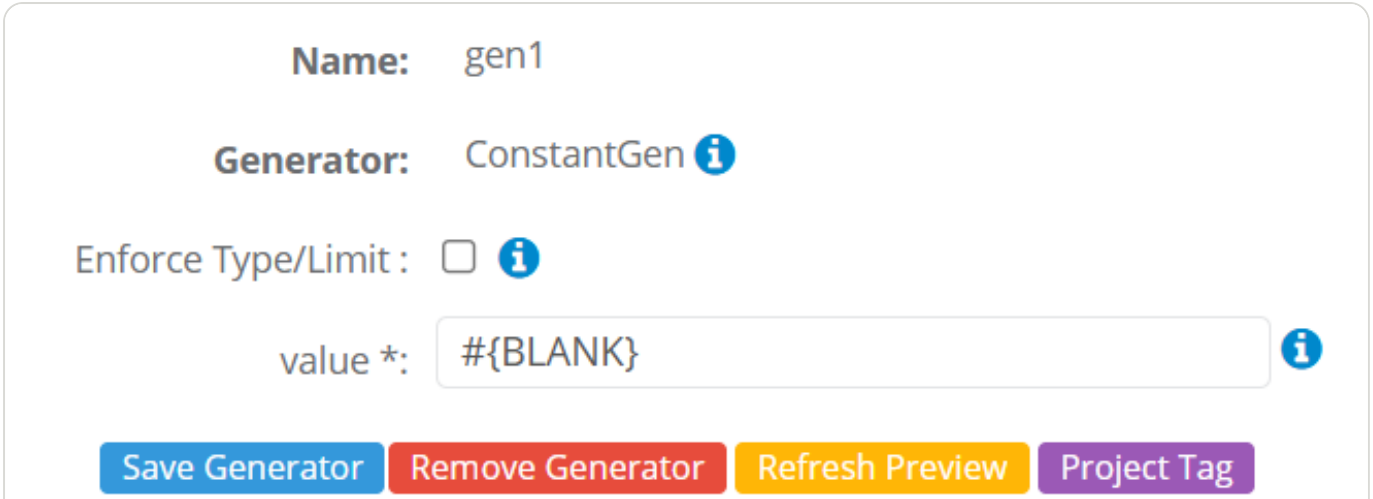
The docs give no worked example of `% Null`; set it, save and watch the Preview pane. It works for any generator because it sits on the attribute.

7.7.3 Place 2: generators that return null or blank

The help centre shows two generators for this purpose, in the walk-through "Flat/Nested JSON: Hide an Attribute when Value is NULL or Blank" (link):

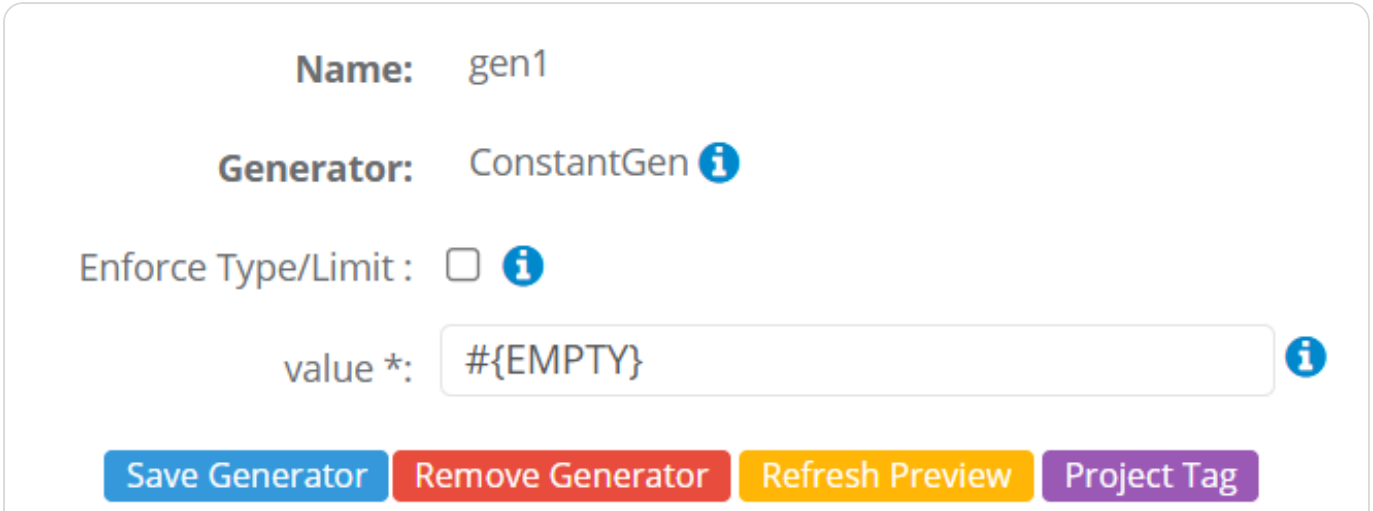
- **NullGen**: "generates a NULL value". The screenshot shows a single parameter, `nullType`, set to NULL. Other `nullType` choices are not documented.
- **ConstantGen**: "generates a constant, non-changing value", and its `value` parameter accepts `#{BLANK}` to generate blank values and `#{EMPTY}` to generate an empty string (ConstantGen).

The string-generators list also names **FillerGen**, a repeating filler value; its parameters are not documented, so check the Preview pane.



Official screenshot: ConstantGen with the `value` parameter set to `#{BLANK}`, the documented way to generate blank values.

`Source page ↗`



Official screenshot: ConstantGen with `value` set to `#{EMPTY}`, the documented way to generate an empty string.

`Source page ↗`

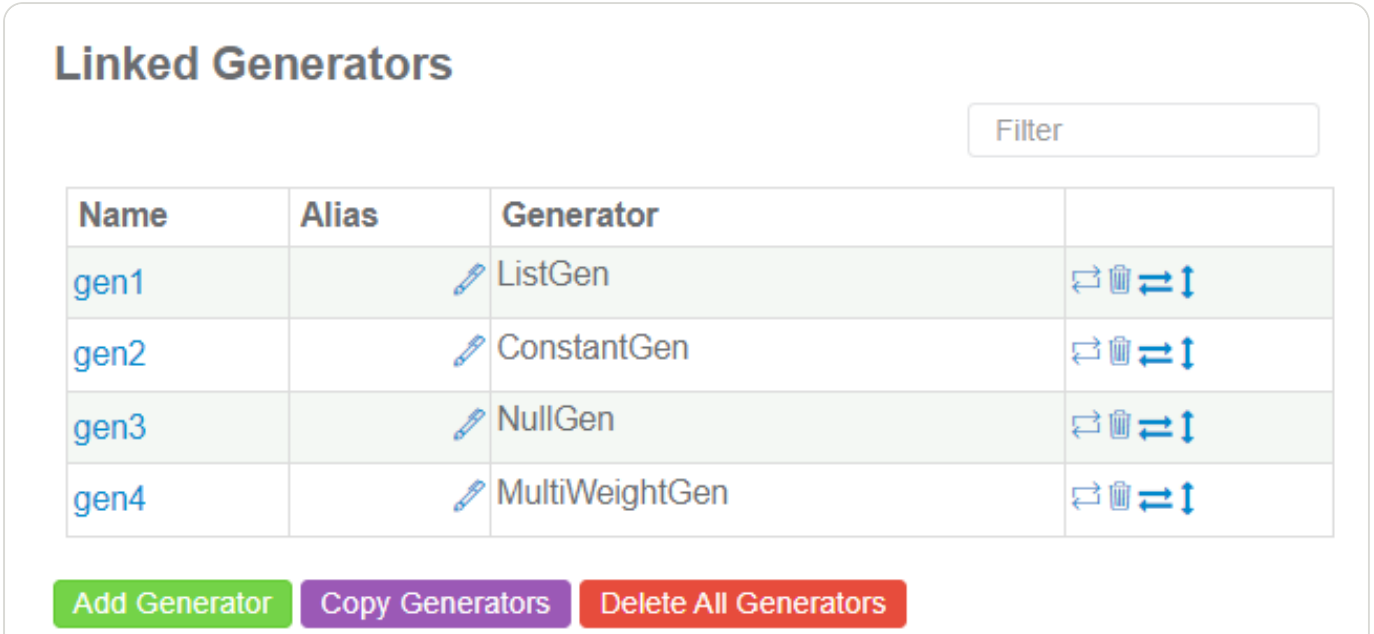
> **BLANK VERSUS EMPTY**
>
> The ConstantGen page lists them as separate examples but does not spell out the difference in the file. Run each in a test scenario and open the output; do not assume.

7.7.4 Mixing normal, null and empty by percentage

The JSON walk-through combines both ideas in one attribute. A tester wants an `address2` (apartment or suite) that is sometimes filled, sometimes null and sometimes empty. Four linked generators do it:

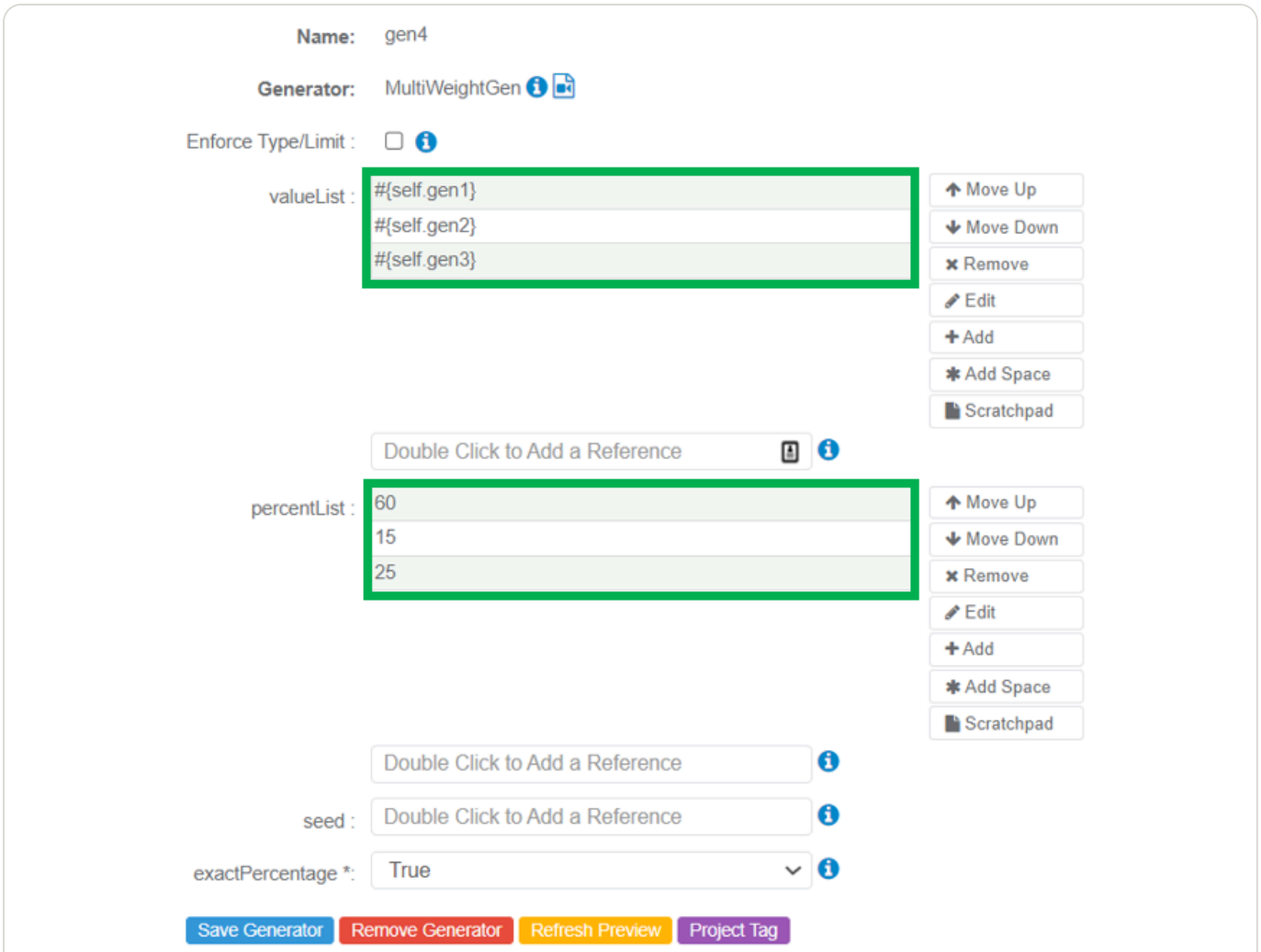
1. `gen1` ListGen: Apt A, Apt B, Suite A, Suite B.
2. `gen2` NullGen: a NULL value.
3. `gen3` ConstantGen with `#{EMPTY}`: an empty value.
4. `gen4` MultiWeightGen with `valueList` `#{self.gen1}`, `#{self.gen2}`, `#{self.gen3}` and `percentList` 60, 15, 25.

About 60 percent of rows get an apartment or suite, 15 percent NULL and 25 percent empty. Only the rightmost generator becomes the attribute's value.



Official screenshot: The four linked generators of the `address2` attribute, in the order `gen1` ListGen, `gen2` ConstantGen, `gen3` NullGen and `gen4` MultiWeightGen. On this screenshot the second row is ConstantGen and the third is NullGen, while the page text (and the separate configuration forms named `gen2` and `gen3`) put NullGen second and ConstantGen third. The page is not consistent with itself, so in your own project always read the Linked Generators table on screen to see which `genN` is which before you wire up the percentages.

`Source page ↗`

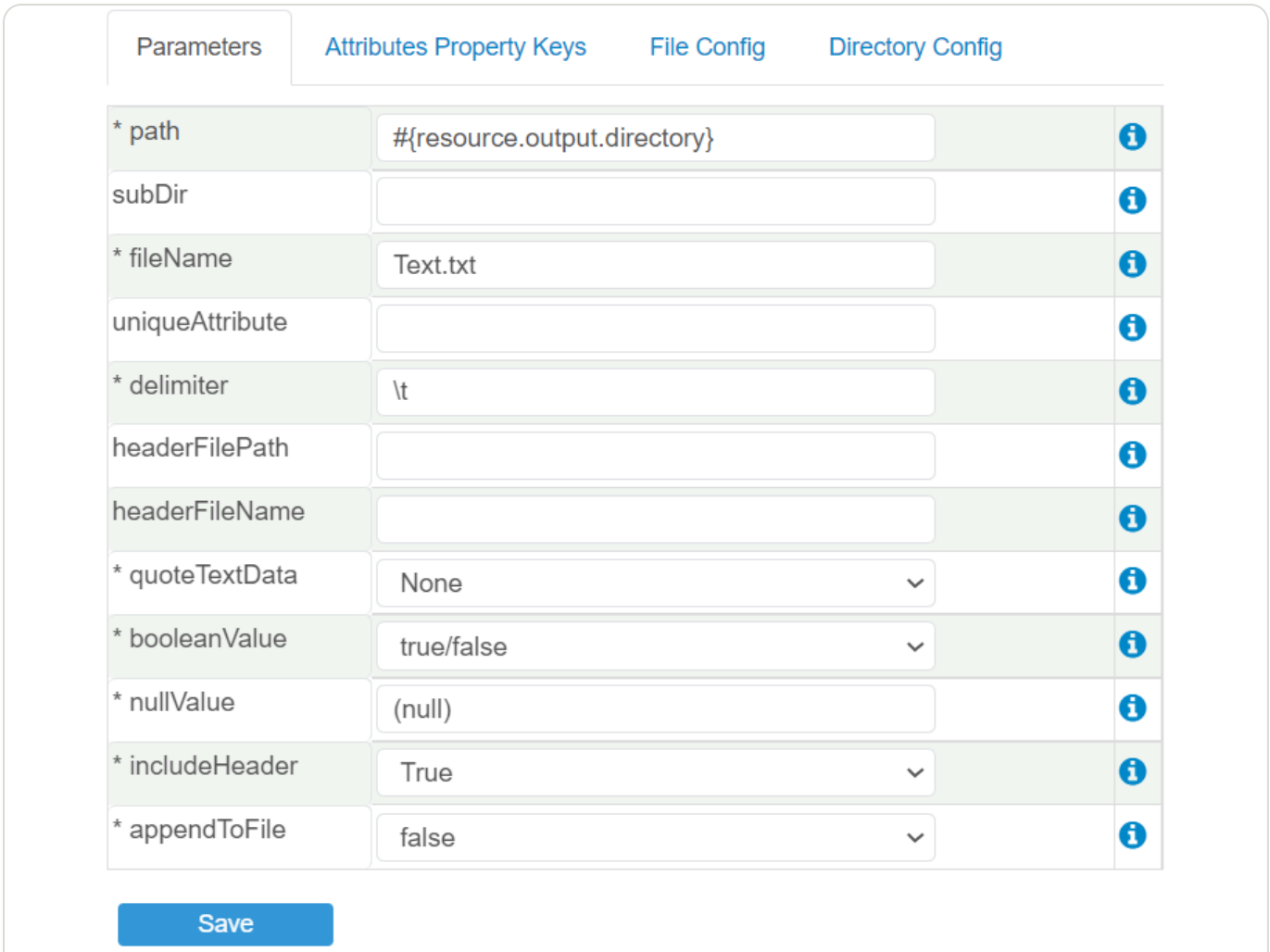


Official screenshot: The MultiWeightGen that mixes the three: `valueList` references `#{self.gen1}`, `#{self.gen2}`, `#{self.gen3}` and `percentList` is 60, 15, 25, with `exactPercentage` True.
`Source page ↗`

Choosing an **Output Condition** on the attribute leaves a null or blank `address2` out of the JSON for that record; the page's sample output omits the key on those rows. That matters for APIs that treat "missing key" and "null value" differently.

7.7.5 Place 3: how the receiver writes a null

A null is only a null inside GenRocket. The receiver decides how it is written to the file. The `DelimitedFileReceiver` documents a required parameter `nullValue` ("Represents a Null value") and a `booleanValue` format (DelimitedFileReceiver). The screenshot of its form shows `nullValue` pre-filled with `(null)`. If a downstream loader expects the literal text NULL or an empty cell, change this parameter before you generate. A mismatch here is a classic source of "the data looks right in GenRocket but my load fails".



Official screenshot: The DelimitedFileReceiver form. Look at `nullValue`, pre-filled with `(null)`, together with `delimiter`, `quoteTextData`, `booleanValue`, `includeHeader` and `appendToFile`.
`Source page ↗`

COMMON NULL-TESTING MISTAKES

- **Only testing NULL.** NULL, empty string and spaces are three different inputs. Cover all three.
- **Nulls in a primary key.** Never use `% Null` on an id or a foreign key unless you are deliberately testing that the database rejects it.
- **Forgetting the receiver.** A null written as `(null)` may load as the five-character text "(null)". Check `nullValue`.

7.8 Boundary values and edge cases

7.8.1 Boundary value analysis, explained from zero

Suppose a form accepts an age from 18 to 65. You cannot test every possible number. Boundary value analysis (BVA) is a way to pick a handful of values that find most bugs. The idea comes from experience: programmers make "off by one" mistakes. They write `age > 18` when they meant `age ⩾ 18`. Such a bug is invisible when you test age 40 and obvious when you test age 18.

Here is the method in four steps.

1. **Find the rule.**
   Write down the allowed range. ShopSphere example: an order total is valid from 0.00 up to 999999.99.

2. **Split the world into groups ("partitions").**
   Everything below the range is invalid, everything inside is valid, everything above is invalid. Testing one value from inside each group covers "the typical case" of each.

3. **Pick the values at each edge.**
   For each boundary test the value just below it, on it and just above it.

4. **Write the expected result for each value.**
   Decide accept or reject for every value before generating anything, so the data file doubles as a test oracle.

For an inclusive range from a minimum to a maximum, the classic set is seven values: just below the minimum, the minimum, just above the minimum, one typical value, just below the maximum, the maximum, and just above the maximum.

BOUNDARY VALUES FOR A SHOPSPHERE ORDER TOTAL VALID FROM 0.00 TO 999999.99

Case	Value	Expected result	Bug it would catch
Just below minimum	-0.01	Reject	App accepts negative totals.
Minimum	0.00	Accept	Code says `total > 0` and rejects a free order.
Just above minimum	0.01	Accept	Rounding drops one cent to zero.
Typical	500.00	Accept	The normal flow itself is broken.
Just below maximum	999999.98	Accept	Off-by-one at the high end.
Maximum	999999.99	Accept	Column too small for the largest legal value.
Just above maximum	1000000.00	Reject	Overflow: the column or code cannot hold it.

An age rule works the same way. Say a ShopSphere customer must be 18 or older on a reference date of 2026-01-01.

BOUNDARY VALUES FOR THE 18-OR-OLDER RULE, REFERENCE DATE 2026-01-01

Date of birth	Age on 2026-01-01	Expected result	Note
2008-01-02	17 (turns 18 the next day)	Reject	The classic 17-versus-18 boundary.
2008-01-01	18 (birthday is the reference date)	Accept	Tests the "on the day" case.
2007-12-31	18 (birthday was one day earlier)	Accept	Just inside the valid group.
1990-04-12	35	Accept	Typical adult (Asha Menon's fake date of birth).

7.8.2 How GenRocket produces boundary data

No boundary generator is documented. The documented building blocks combine into a few reliable recipes.

Recipes for boundary and edge data, all built from documented generators

Need	Recipe	Source page
A small fixed set of exact values, each exactly once	ListGen with `setLoop` True and the values in `list`. The loop count becomes the list length (up to 15 items).	ListGen
Mostly typical data, with the edge value every few rows	EdgeCaseGen with `source` the typical generator, `edgeCase` the edge value and `cycle` N.	EdgeCaseGen
Mostly typical data, with an edge value a set share of the time	EdgeCaseGen with `percentage`, or WeightGen/MultiWeightGen.	EdgeCaseGen; WeightGen
Too-long text for a column	EdgeCaseGen with a list of short names and a list of long names, or RandomStringGen with a large `length`.	EdgeCaseGen; RandomStringGen

Why exact lists: a random generator might hit 999999.99 once in a million rows, or never; a boundary test must hit it every time.

7.8.3 Lab: a boundary-value suite for ShopSphere orders

Lab: generate the seven order-total boundary rows

Goal: a small domain that produces exactly one row per boundary case, with the expected result next to each value, and a delimited file you can hand to a tester. This lab reuses the project setup steps from the 50-record lab. The lockstep behaviour of three ListGens in one domain is an inference; confirm it in the Preview pane.

1. **Create a domain named `OrderBoundary`.**
   In the Domains pane choose New Domain, then New Domain, enter the name and Save. Domain names take letters and numbers only. The new domain starts with only an `id` attribute.

   Expected result: A Domain Dashboard with `id` and the variable `global.OrderBoundary.loopCount` equal to 1.

2. **Add attribute `caseName` and give it a ListGen.**
   New Attribute, name it, Save. Replace the default generator with ListGen (Quick Generator Replacement menu, type ListGen, Replace). In `list` enter one item per line: below min, min, min plus 1, typical, max minus 1, max, above max. Set `setLoop` to True.

   Expected result: The Preview shows seven labels. In a scenario the loop count will be locked to "N", as the ListGen page describes.

3. **Add attribute `totalAmount` with a ListGen.**
   Enter -0.01, 0.00, 0.01, 500.00, 999999.98, 999999.99, 1000000.00, one per line, in the same order. Leave `random` False and `setLoop` False.

   Expected result: The Preview lists the seven amounts in order. If a trailing zero such as 0.00 is shown as 0, see the formatting note below.

4. **Add attribute `expected` with a ListGen.**
   Enter REJECT, ACCEPT, ACCEPT, ACCEPT, ACCEPT, ACCEPT, REJECT in the same order.

Expected result: Rows line up: row 1 is "below min", -0.01, REJECT; row 7 is "above max", 1000000.00, REJECT.

5 **Add a DelimitedFileReceiver, create a scenario and run it.**
Follow steps 11, 12 and 14 to 16 of the 50-record lab: add the receiver with a comma delimiter and a file name such as `OrderBoundary.csv`, create a scenario with this domain as the primary domain, download it and run `genrocket -r OrderBoundary.grs` (use your scenario's name).

Expected result: A file with a header and exactly seven rows. A row count other than seven means `setLoop` was not applied to the first generator of `caseName`.

▶ **Show a worked answer**

7.9 Enforcing data type and size

7.9.1 In plain words

Every database column has a type and a size. A `first_name VARCHAR(5)` column holds at most five characters. An `INT` column holds whole numbers up to about two billion. If your generator produces "Maximiliano" for that first-name column, the insert fails, and a hundred rows of perfectly good data are lost because one name was too long.

For *positive* data, generated values must always fit the column, so loads never fail; GenRocket does this with a switch called Enforce Type/Limit. For *negative* data (the long-name edge case in section 7.6.5) you deliberately break the size.

7.9.2 Enforce Type/Limit

The page "How to Enforce Data Type and Size in Generated Data" explains the option (link):

- Value limits are enforced automatically when you import a DDL (database table definition) or an XTS file (GenRocket's extracted table schema). The option is switched on for every generator assigned during such an import.
- It is available on every generator and can be set on any individual generator in a domain.
- Use it "whenever you want the generated data to be within a defined data type and size".

- When on, it limits output to the `Type` and `Size` parameters of the attribute.

The page's example: NameGen with Enforce Type/Limit on, `Type` varchar and `Size` 5. First names in the Preview pane are limited to 5 characters.

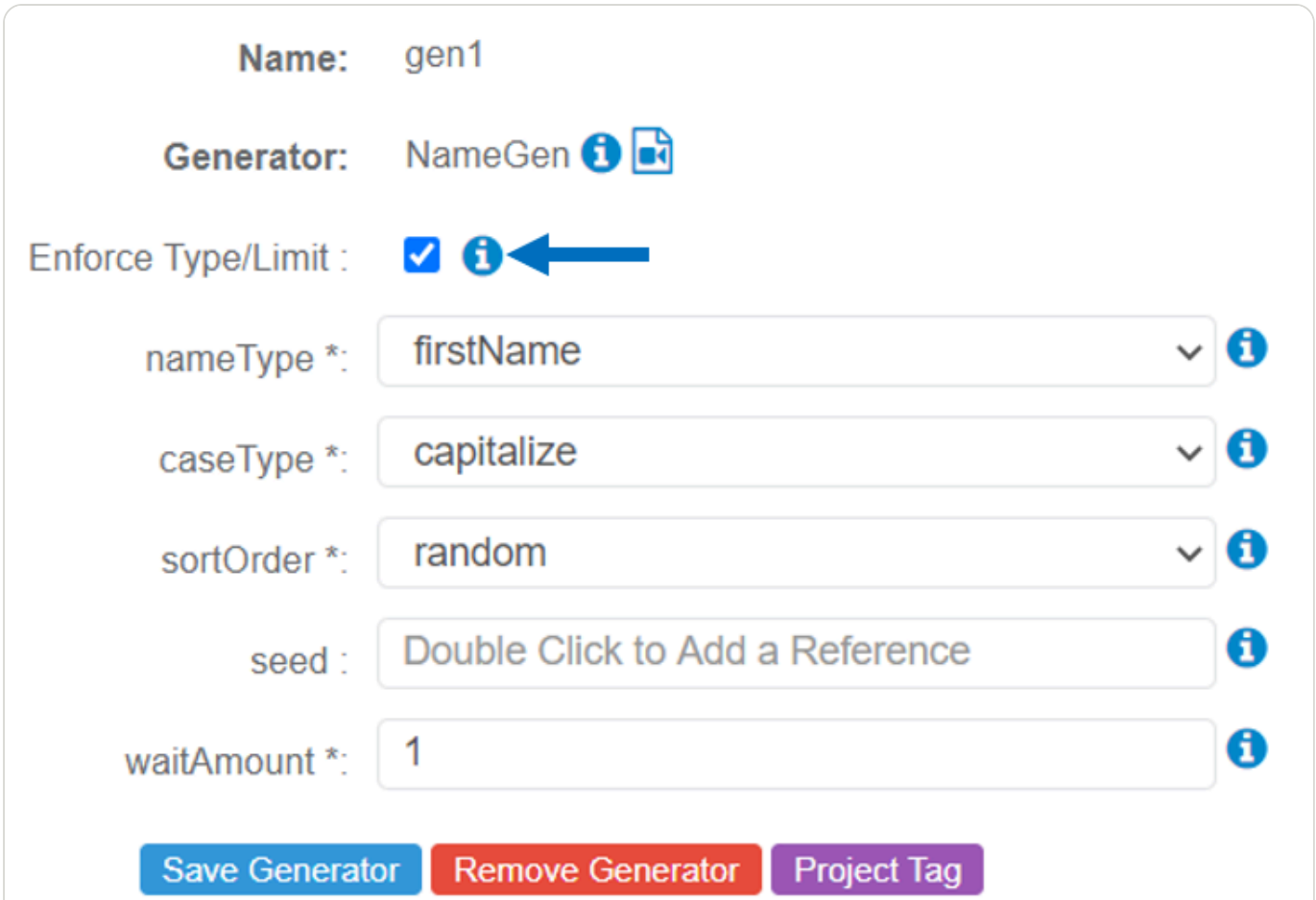
Integer limits the page lists for Enforce Type/Limit

Data type	Largest value enforced
`tinyint`	127
`smallint`	32767
`mediumint`	8388607
`int` or `integer`	2147483647
`bit` or `vbit`	1

These are boundary values in their own right: for a `smallint` column, 32767 is the maximum and 32768 is the "just above maximum" negative test.

7.9.3 Setting it by hand

For a manually built domain: open the attribute, type a data type (`varchar` , `int` , `float` , `decimal`) in **Type** and the limit in **Size**, click Save, then tick **Enforce Type/Limit** in the generator form and save the generator. The Preview pane then shows values within the size.



Official screenshot: The NameGen form with the **Enforce Type/Limit** checkbox ticked (arrow), nameType firstName, caseType capitalize and sortOrder random.

`Source page ↗`

> **TURNING IT OFF FOR EDGE CASES**
>
> Because it is on by default for imported models, a generator will not produce the too-long name a negative test wants. Switch it off on that generator, or use a separate generator for the edge value, and check Preview.

7.9.4 Decimal places and number formatting

Money needs fixed decimal places: 49.9 and 49.90 are different text, and a `DECIMAL(8,2)` column expects two.

NumberFormatGen formats a referenced number: pad with leading zeros, fix the number of decimal places, fix the number of digits, add commas, include a sign. Parameters (NumberFormatGen):

- `reference*`: the attribute (or linked generator) to format. It should produce a floating-point number.

- `decimalFormat*`: a pattern. The page's sample: `123456.789` with `###.##` gives 123456.79.
- `includeSign*`, `includeDecimal*` and `padLength*`: whether a sign and the decimal point appear, and how many zeros to pad on the left.

The page links NumberFormatGen to **RangeDecimalGen** and **RandomDecimalGen**, which supply the decimal number. Those two have no separate pages; from the page's screenshots, RangeDecimalGen shows parameters `startRange`, `endRange`, `decimalFormat`, `jumpAmount`, `waitAmount` and `sortOrder`, and RandomDecimalGen shows `startRange`, `endRange`, `decimalFormat` and `seed`.

gen1	gen2
-10.00	-00010.00
-9.25	-00009.25
-8.50	-00008.50
-7.75	-00007.75
-7.00	-00007.00
-6.25	-00006.25
-5.50	-00005.50
-4.75	-00004.75
-4.00	-00004.00

Official screenshot: The output of the page's first example: `gen1` is a decimal sequence from -10.00 in steps of 0.75, and `gen2` is the same number padded to five digits with the sign kept (-00010.00, -00009.25 and so on).

Source page ↗

gen1	gen2
53.8970	53.90
49.6710	49.67
72.9353	72.94
92.5710	92.57
95.7306	95.73
38.7472	38.75
60.4947	60.49
23.8590	23.86
23.2817	23.28

Official screenshot: The result of formatting those numbers to two places: `gen1` shows 53.8970 and `gen2` shows 53.90, 49.6710 and 49.67, and so on. This is the pattern for money columns.

Source page ↗

For the typical partition of `total_amount` use RandomDecimalGen from 1 to 5000 with a `seed`; by analogy with the screenshot's `#0.0000`, two decimals would be `#0.00` (check in Preview). Boundary rows come from the boundary lab.

7.10 Dates and times

7.10.1 In plain words

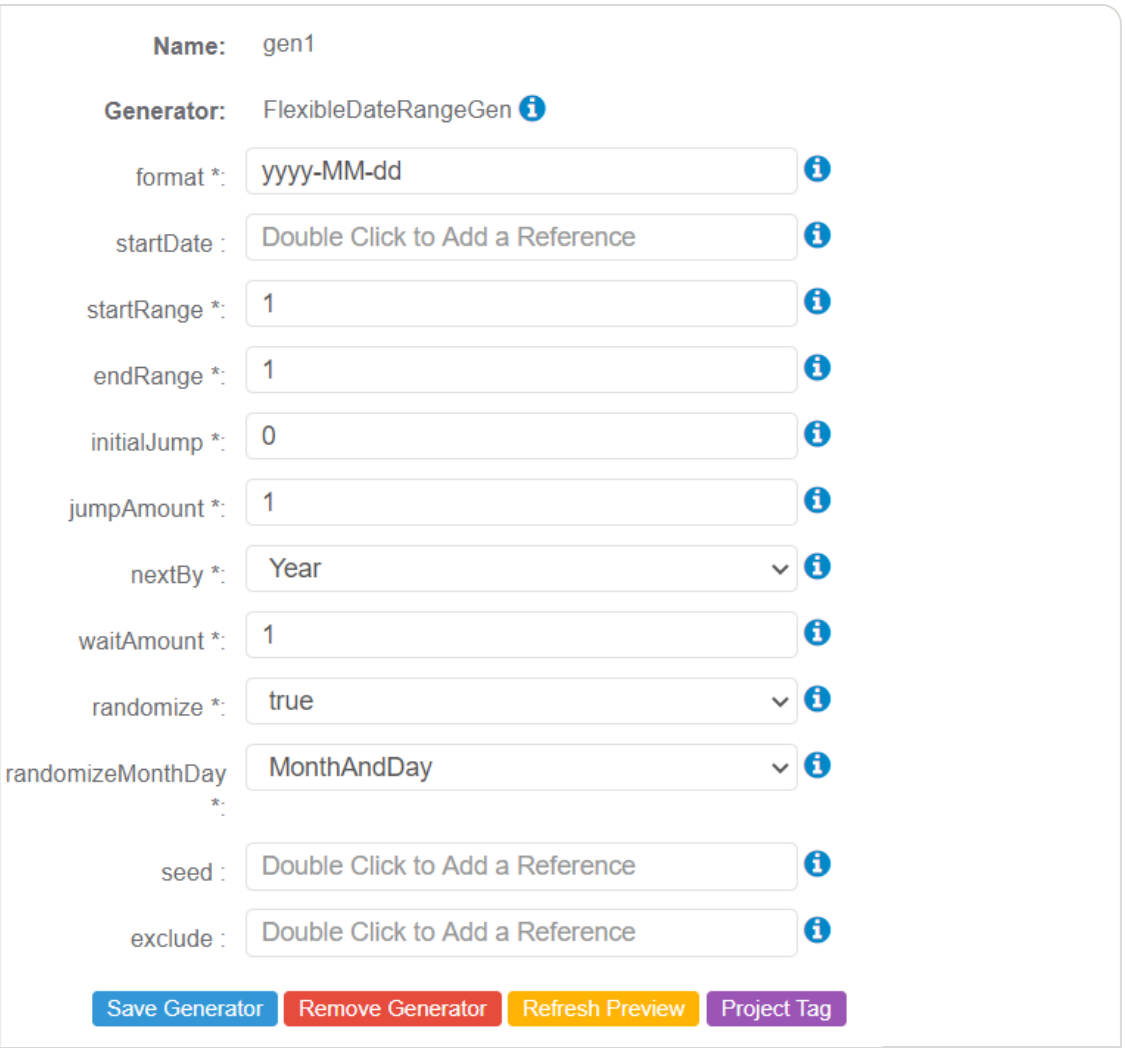
Dates breed bugs: a birth date that makes someone 17 versus 18, an order dated 29 February, a timestamp at midnight, month-first versus day-first formats. Date test data needs three things: a *format* (how the date is written), a *range* (earliest and latest) and a *pattern* (sequential, random, or specific edge dates).

7.10.2 FlexibleDateRangeGen

GenRocket automatically picks **FlexibleDateRangeGen** for attributes whose imported data type is "date" or "time" (How are Generators Assigned). It generates dates from a start date, moving by a range and a jump (FlexibleDateRangeGen).

FlexibleDateRangeGen parameters as documented

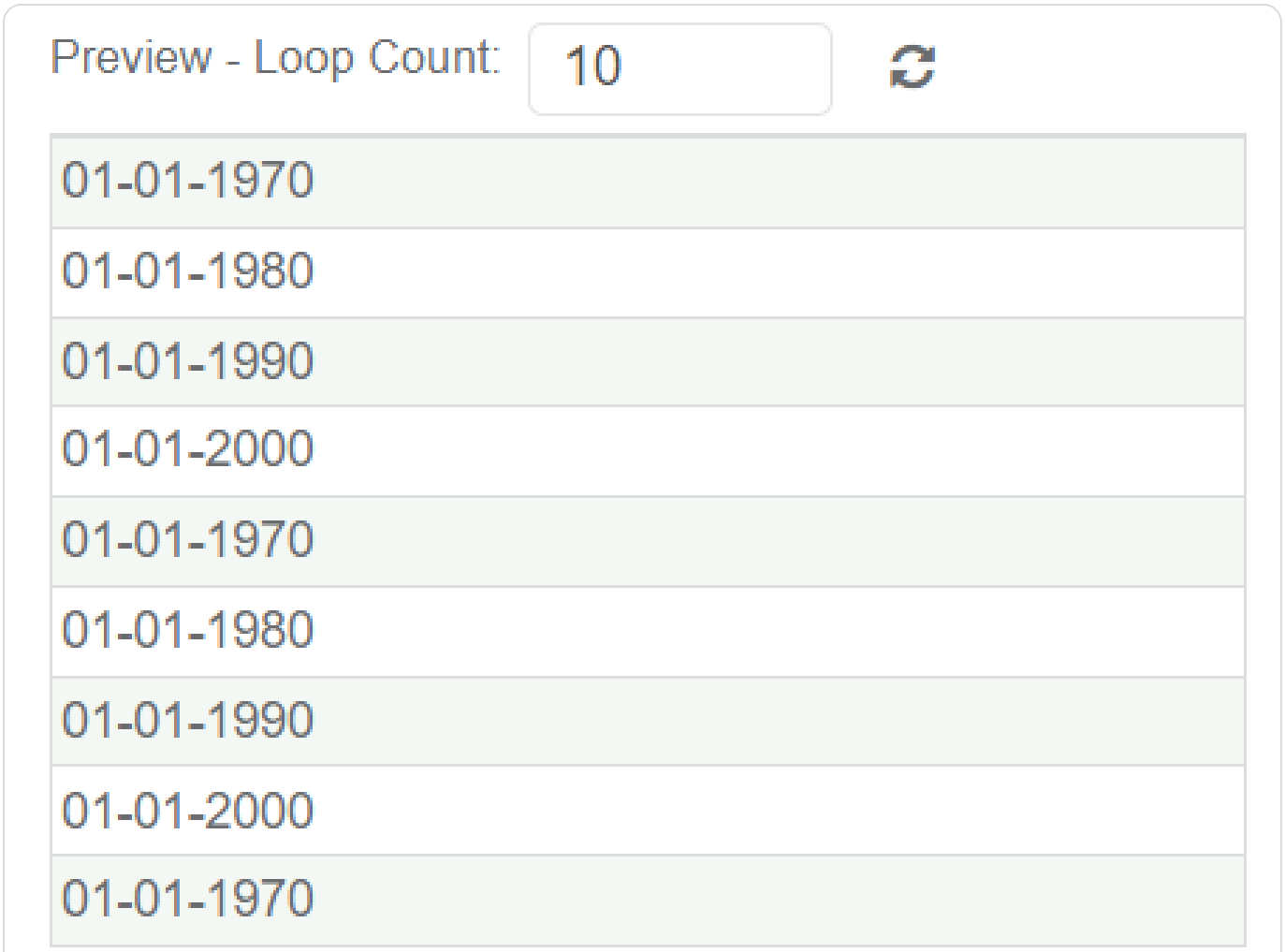
Parameter	Meaning
`format*`	The date format for the start date and the output. Default `yyyy-MM-dd`. Others: `MM/dd/yy`, `MM-dd-yy`, `yyyy-MM-dd HH:mm:ss`.
`startDate`	The anchor date. Empty means the current date.
`startRange*`, `endRange*`	Integers: how far forward (positive) or backward (negative) to move from the start date. Zero means no change. An error occurs if `startRange` is more than `endRange`.
`initialJump*`	How much to jump before the first value (Year, Month, Week, Day or Hour).
`jumpAmount*`	How much to jump by for each value.
`nextBy*`	Which part of the date moves: Year, Month, Day, Hour, Minute, Second or Millisecond.
`waitAmount*`	Rows to repeat a date before moving on.
`randomize*`	Sequential (false) or random (true) within the range.
`seed`, `randomizeMonthDay*`, `exclude`	Repeatable random dates; whether month and day are random; values to skip.



Official screenshot: The complete FlexibleDateRangeGen form on one screen: `format`, `startDate`, `startRange`, `endRange`, `initialJump`, `jumpAmount`, `nextBy`, `waitAmount`, `randomize`, `randomizeMonthDay`, `seed` and `exclude`.

Source page ↗

The page's first example: `startDate` 01-01-1970, `startRange` 0, `endRange` 4, `initialJump` 0, `jumpAmount` 10, `nextBy` Year, `randomize` false and `randomizeMonthDay` None. The first record is 01-01-1970 and the year grows by 10 each record: 1970, 1980, 1990, 2000, then the pattern restarts at 1970 on the fifth record.

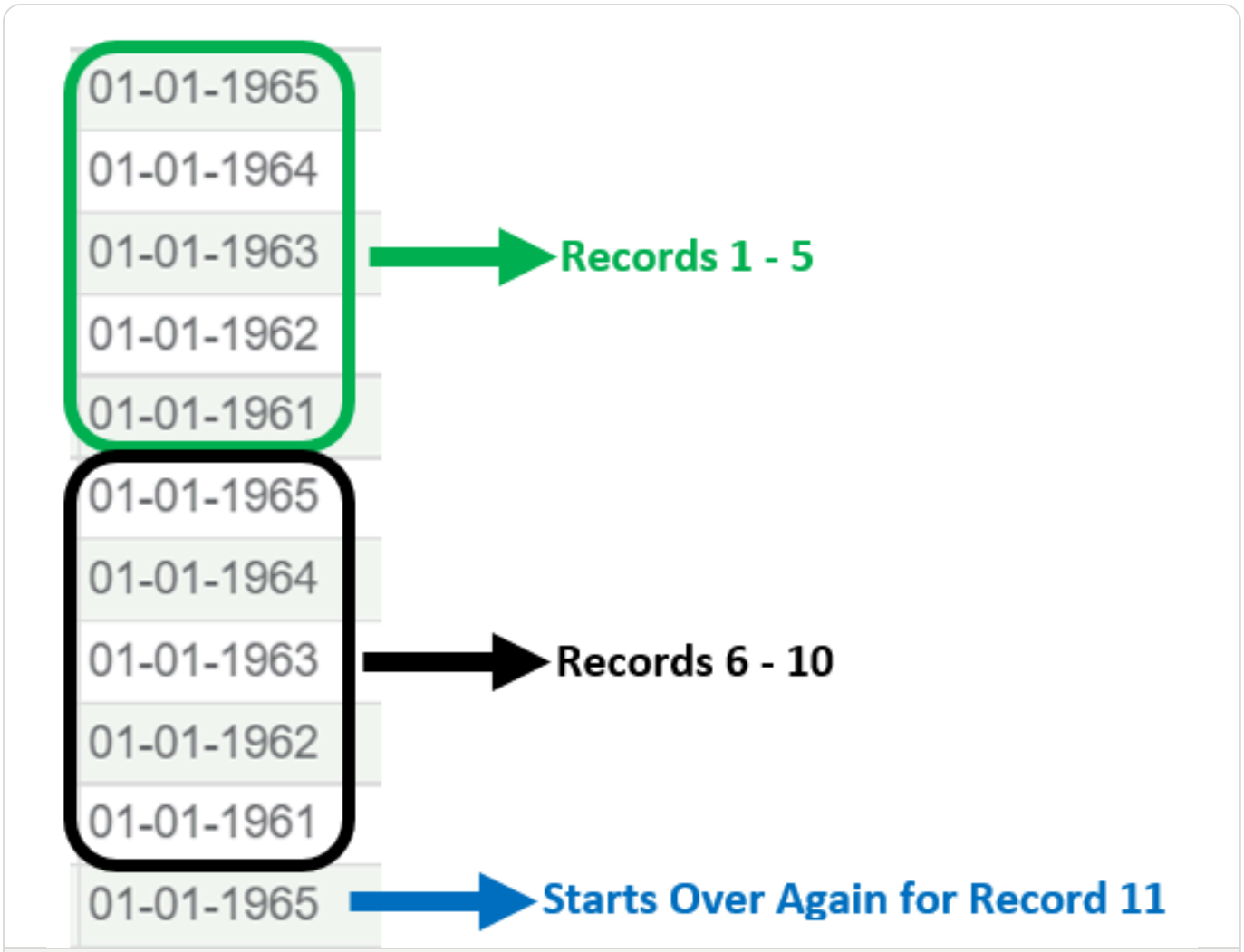


Official screenshot: The preview of that example: 01-01-1970, 01-01-1980, 01-01-1990, 01-01-2000, then 01-01-1970 again. The four-record cycle comes from `endRange` minus `startRange`.

Source page ↗

CHECK THE DIRECTION YOURSELF

Example 2 contradicts itself: its `jumpBy` line says the year increases, but the title, records and screenshot go 1965 down to 1961. In the examples a cycle holds `endRange` minus `startRange` values (0 to 4 gives four dates). Change one parameter at a time and read the Preview pane.



Official screenshot: The preview for the negative-range example: 01-01-1965, 01-01-1964, 01-01-1963, 01-01-1962, 01-01-1961, then back to 01-01-1965. The decreasing order is what the screenshot shows.
`Source page ↗`

7.10.3 Date formats

The "Date and Time format" page lists the pattern letters (link). The ones you will use most:

Year is `y`, month `M`, day `d`, hour `H` (0 to 23) or `h` (1 to 12), minute `m`, second `s` and am/pm `a`.

Capital `M` is month and lower-case `m` is minute, so `yyyy-mm-dd` prints the minute where you wanted the month. Attributes also have a Date Format parameter that "defines the Date Format if the Attribute is generating a Date".

7.10.4 DateDeltaGen: the gap between two dates

DateDeltaGen "computes the delta between two dates", in days, months, years or hours. Parameters: `conversion*`, `formatOne*`, `formatTwo*`, `dateRefOne*`, `dateRefTwo*`. The page's example links two FlexibleDateRangeGen generators and subtracts their dates (DateDeltaGen).

That gives a way to check an age rule in the data itself: generate a `date_of_birth`, hold a reference date in a ConstantGen (the ConstantGen page lists dates such as an effective date as a valid constant), and use DateDeltaGen with `conversion` years between them. The pages do not say how partial years are rounded, so verify the 17 and 18 rows by eye.

WORKED EXAMPLE: THE 17 VERSUS 18 ROWS

For the boundary rows, a ListGen with `setLoop` True holds `2008-01-02`, `2008-01-01`, `2007-12-31` and `1990-04-12`, the four rows of the age table in section 7.8.1. For a realistic spread of adult birth dates use FlexibleDateRangeGen with `nextBy` Year, `randomize` true and a `seed`, and check the range cannot produce under-18 dates unless you want them.

7.11 Pattern coverage table

Each row is a pattern, how GenRocket produces it, and the documented generator and parameters to reach for first.

Data patterns, GenRocket mechanisms and ShopSphere examples

Pattern	GenRocket mechanism	Generators and parameters (documented)	ShopSphere example
Sequential id	Count up	Domain `id` attribute; RangeGen `startRange`, `jumpAmount`, `endRange` empty	`order_id` 5001, 5002 ...
Continue after existing rows	Linked query plus range	QueryOnceGen `resourceName`, `query`, `sampleValue`; RangeGen `startRange` = `#{self.gen1}`	Insert into a non-empty table
Sequential phone numbers	Sequence generator	PhoneNumberGen `areaCode`, `prefix`	(800) 600-1000 onward
Random number	Random in range	RandomGen `startRange`, `endRange`, `uniqueLimit`, `seed`	Quantity 1 to 10
Random names	Random from a name pool	NameGen `nameType`, `caseType`, `sortOrder` random, `seed`	Asha, Menon
Repeatable random data	Seed	`seed` on the generator; domain Global Seed variable	Same customers every night
Weighted values	Percentage choice	WeightGen `percentage`, `variable1`, `variable2`; MultiWeightGen `valueList`, `percentList`, `exactPercentage`	70 percent Delivered
Conditional value	If/else on another value	EvalCaseGen `equation`, `var1`, `caseTrue`, `caseFalse`; SwitchGen; IfThenElseGen	age >= 18 gives Yes
Edge case now and then	Mix normal and edge value	EdgeCaseGen `source`, `edgeCase`, `cycle` / `percentage` / `condition`	Negative age; 12-letter name

Pattern	GenRocket mechanism	Generators and parameters (documented)	ShopSphere example
Exact boundary values	Fixed list, one row each	ListGen `list`, `setLoop` True	0.00, 999999.99
All combinations	Permutations	PermutationGen `list`	Size by colour by gift wrap
Null values	Attribute null settings	Attribute `% Null`, `Null Value`; NullGen `nullType`	10 percent null email
Blank or empty values	Constant generator	ConstantGen `value` `#{BLANK}` or `#{EMPTY}`	Empty email
Fit the column	Enforce Type/Limit	Attribute `Type`, `Size`; the checkbox	`varchar(5)` names
Dates in a range	Date range	FlexibleDateRangeGen `format`, `startDate`, `nextBy`, `randomize`	Birth dates

7.12 Common mistakes and gotchas

MESSAGES AND MISTAKES THE DOCUMENTATION NAMES

Symptom	Cause	Fix
"Exhausted attempts to find the next valid random value"	RandomGen `uniqueLimit` too small for the loop count	Raise `loopCount`, `endRange` or `uniqueLimit`
"startRange cannot be less than or equal to endRange"	Descending RangeGen with a positive end	Use a negative `endRange` or ascending order
ListGen throws when `random` and `unique` are True	Loop count larger than the list	Shorten the loop count or lengthen the list; the Preview cannot show this case
Referenced value comes out empty or null	The referenced attribute or generator is below or to the right	Rearrange Attributes; reorder linked generators

7.13 Recap

RECAP

- Test data needs patterns: typical data for positive tests, tricky data (boundary, edge, null) for negative tests.
- Sequential: RangeGen, the domain `id`, ListGen in order, PhoneNumberGen. A range that wraps is not unique.
- Random: RandomGen, NameGen, ListGen and RandomStringGen; a `seed` makes random data repeatable, and equal seeds repeat each other.
- Weights: WeightGen for two-way splits, MultiWeightGen with `exactPercentage` True for exact shares.
- Linked and conditional generators run left to right and may only reference what is to their left (and attributes above them). EvalCaseGen handles if/else; EdgeCaseGen mixes in edge values by cycle, percentage or condition.
- Null and blank: attribute `% Null`, NullGen, ConstantGen with `#{BLANK}` or `#{EMPTY}`, Output Condition, and the receiver's `nullValue`.
- Boundary value analysis: test just below, on and just above each limit, plus one typical value. In GenRocket use exact lists (ListGen with `setLoop` True) for boundaries.
- Enforce Type/Limit keeps values inside the column; NumberFormatGen fixes decimals; FlexibleDateRangeGen and DateDeltaGen cover dates.
- The vendor's reason for patterns (its videos): masked copies lack negative, permutation and stateful data, and random data is hard to test with.

Hands-on: generate 50 customer records with GenRocket

This section generates **50 records with the columns FirstName, LastName, Full Name (FirstName MiddleName LastName) and Phone Number** in GenRocket, click by click, with the reason for each click and what should be seen. It uses the vocabulary of the GenRocket overview, domains, attributes and generators and receivers and scenarios, and the patterns of the data patterns chapter.

A short section on planning test data generation end to end comes first. Real work never starts at the keyboard: someone decides what data is needed, tries a small proof first, checks the environment, and plans how generation will run every week. The 50-record lab is, in miniature, the proof-of-concept step of that plan.

7.14 Words to know first

TDG (test data generation)

Creating test data from scratch with a tool instead of copying production. Here it means doing it with GenRocket.

POC (proof of concept)

A small, quick trial that shows an approach can work before you commit to the full effort.

Loop count

How many rows a domain generates. The documentation variable is `global.<DomainName>.loopCount` and it starts at 1.

Reference

A parameter value that points at another attribute or generator, written like `#{self.gen1}` or `#{Domain.attribute}`, so one value is built from others.

Visible

An attribute setting. Visible means it is written to the output; not visible means it is used only as a helper.

Delimited file

A text file where each row is one record and columns are separated by a character such as a comma or tab. CSV is the comma version.

Output directory

The folder where files are written. GenRocket stores it as an organization resource named `resource.output.directory`, set per user.

User profile

A small file downloaded from GenRocket Cloud that links your computer to your licence. It goes in a `.genrocket` folder and is needed to run a scenario.

7.15 Planning test data generation end to end

A TDG plan answers four questions in order. Each stage ends with something you can show, so a reviewer can say "go" or "stop".

1. Use cases what data, for which tests → **2. Proof of concept** small trial → **3. Assessment** application and data environment → **4. Deployment plan** run it every week

7.15.1 Stage 1: identify the TDG use cases

A use case is a reason someone needs generated data. Ask each team what they test and what breaks today.

SHOPSPHERE TDG USE CASES (EXAMPLE DATA)

Use case	Who needs it	Data needed	Pattern (see the patterns chapter)
Unit and API tests for checkout	Developers	Small, exact, repeatable customer and order rows	Sequential keys, seeded random
Negative tests for validation	QA	Empty emails, too-long names, 0.00 and 999999.99 totals	Null, blank, boundary, edge
Regression pack	QA	Same data every night	Seeds, fixed lists
Load test	Performanc e team	Very many rows, valid relationships	Large loop counts, parent and child
Demo or training environment	Support	Realistic but fake customers	Random names, weighted values

GenRocket's CI/CD page lists four facts you need before you can generate anything: the *object* (for example User or Customer), the *properties* of it (name, phone number), *what data* each property should hold, and the *format* to deliver it in (CSV, JSON, XML) (Automating Data Generation using

CI/CD Pipeline). Those map straight onto GenRocket's domain, attributes, generators and receiver. Collect those four answers for every use case. In the lab: object Customer, properties FirstName, LastName, FullName and PhoneNumber, generators NameGen, ConcatGen and PhoneNumberGen, format CSV.

7.15.2 Stage 2: the proof of concept

A POC picks one narrow use case and builds it end to end, so you learn the tool and find surprises early. A good POC:

- Uses **one domain**, a few attributes and a small row count (the 50-record lab fits).
- Covers the **whole path**: model, design, deliver, run, check the file. A half-built POC proves nothing.
- States **success criteria** in advance, for example "50 rows, no empty values, phone numbers all the same length, runs in a command line without manual edits".
- Records **what was hard**: licence, runtime install, access to the output folder.

The documentation frames GenRocket's own method as Model, Design, Deploy, Manage (The GenRocket Methodology, and the CI/CD page above spells the four steps out); the concepts chapter gives what GenRocket's 2022 video says each step means. The POC exercises the first three: you model a Customer domain, design its attributes and scenario, and deploy by running the scenario. "Manage" is stage 4.

7.15.3 Stage 3: assess the application and data environment

Before you scale up, look at what is around the data. The table gives the questions and the documented facts that help you answer them.

Environment assessment checklist

Question	Why it matters	Documented anchor
Where is the data model defined?	If a DDL, XSD, CSV, JSON or YAML exists, import it instead of typing attributes.	The project-creation page lists import options: XTS, XSD, CSV, DDL, JSON, JSON Schema, Avro JSON Schema, Spark Schema, YAML, Salesforce and Presets; or create domains manually (link).
What formats must the data arrive in?	Decides the receiver.	"120+ types of Receivers" (receivers overview). We use DelimitedFileReceiver.
How are tables related?	Parent and child domains keep referential integrity.	Each domain can have one parent and many siblings (link).
How many rows, how often?	Sets loop counts and hardware.	Runtime page recommends 4 CPUs and 8 GB RAM minimum (16 GB recommended), and a large heap for large uniqueness sets (Runtime Overview).
Where will the Runtime run?	Local machine, shared machine or server.	Runtime Overview lists those three, plus Docker.
Which Java version is installed?	Runtime needs a supported one.	Runtime Overview: 8 (except 1.8u20), 11, 17, 21, 24. Check with `java -version`.
Can the machine reach GenRocket for licence checks?	Setup "requires access to the firewall for license verification".	Runtime Overview.

Question	Why it matters	Documented anchor
Is any real data involved?	Synthetic data contains none, which removes the masking step.	Why Synthetic Test Data page.

7.15.4 Stage 4: the deployment plan

The deployment plan says how generation becomes routine.

- **Roles.** The documentation names the "Power User, a Test Data Engineer (or TDE)" as the start of the workflow (Test Data Workflow). Decide who models, who approves, who runs.
- **Where scenarios live.** Manual download from the web platform, or G-Repository, which the G-Cases page calls "Best Practice" because users do not download files.
- **How it is triggered.** The CI/CD page says the Runtime can be triggered "from within CI/CD pipeline script using shell or batch command", or through a REST request to GMUS (GenRocket Multi-User Server).
- **Versions and changes.** Project versions hold each release's model. When the application's tables change, someone updates the domains.
- **Data variety on demand.** Test Data Cases (G-Cases) change volume and variety without editing scenarios.
- **Exit criteria.** Define what "done" is: all use cases served, runbook written, owners named.

> **WHY PLAN BEFORE YOU CLICK**
>
> Teams that start with the tool often generate data nobody asked for, in a format nobody can load. A one-page plan with the four stages prevents that.

7.16 What we are building, and the design decisions

The target is four columns and 50 rows. Each column's source is below.

TARGET FILE LAYOUT AND HOW EACH COLUMN IS PRODUCED

Column header	Attribute	Generator	Notes
(not in output)	`id`	automatic	Every domain gets an `id` attribute. We exclude it from the file with the receiver's `include` setting.
FirstName	`firstName`	NameGen, `nameType` firstName	GenRocket assigns NameGen automatically to attributes with "name" in the name.
(not in output)	`middleName`	NameGen	Helper only. `Visible` is switched off.
LastName	`lastName`	NameGen, `nameType` lastName	
Full Name	`fullName`	ConcatGen	First name, a space, middle name, a space, last name.
Phone Number	`phoneNumber`	PhoneNumberGen	Sequential numbers, one per row.

Four choices deserve a reason.

1. **Why a separate `middleName` attribute?** Full Name is defined as "FirstName MiddleName LastName". For the Full Name to contain the same first and last name as the other columns, it must *refer to* those columns, not generate new names. So the middle name needs its own attribute too.
2. **Which generator for the middle name?** NameGen's documented `nameType` choices are firstName, lastName, maleName and femaleName. There is no "middleName" type, so `firstName` is used, because middle names are usually given names (a choice, not a documented rule). The help centre also has a **FullNameGen**, which generates "a full name consisting of a first, middle, and last name with an optional title and suffix" (FullNameGen). It is a valid alternative, but it creates its own first and last names inside one value, so they would not match the FirstName and LastName columns. Its title and suffix percentages also default to 100, so you would have to set them to 0.

3. **Why reference attributes instead of linked generators?** The documentation shows both. The *linked* way builds the full name inside one attribute with three generators and no helper attributes (Linked Generators Overview, "Generate a Person's Full Name"). Our task needs separate FirstName and LastName columns as well, so we build the pieces as real attributes and reference them. The overview notes the cost of this approach: the helper attributes "would then need to have the visibility Parameter set to false"; otherwise they appear in the output.
4. **Why PhoneNumberGen and not a random phone generator?** PhoneNumberGen is documented, with parameters. A random alternative, RandomPhoneNumberGen, is named on its page and appears in an API example with the parameters `useAreaCode` and `uniqueNumbers`, but it has no page of its own, so the main lab does not use it.

> **WHAT THE DOCS DO NOT SAY ABOUT PHONE FORMAT**
>
> The PhoneNumberGen page describes a sequence "100-1000, 100-1001, ..., 999-9999" and its preview screenshot shows numbers like `(800) 600-1000`. It does not list a format option. If your project needs 10 plain digits like ShopSphere's sample phone `9876543210`, the documented route is a second linked generator such as StringRemoveGen (parameters `reference`, `valueToRemove`, `removalType`) or StringReplacementGen (`value`, `fromText`, `toText`) to strip punctuation. See the extension ideas in section 7.22.

7.17 Before you start: the checklist

The lab has two halves. The **design half** happens in the GenRocket web platform in your browser. The **run half** happens on a computer with the GenRocket Runtime installed (see the overview chapter for what the Runtime is). You need both.

- **A GenRocket login** with permission to create projects. If the New Project button is missing, the project page says you lack that permission and must ask the organization admin.
- **The Runtime installed** and working: the command `genrocket` must be recognised in a terminal. The setup page for your operating system covers it, along with a supported Java version.
- **Your resource values set.** Each user sets the path for `resource.output.directory` (where output files are written) and `resource.home.directory` in the upper right of the Project Dashboard. The help centre recommends using different folders for output and home, and not the same path for all (resource values and your user profile).
- **Your user profile downloaded** and placed in the `.genrocket` folder. Every time you change a resource value, download it again (Download menu, Download My Profile).

- **A text editor or spreadsheet** to open the result.

7.18 Lab: generate 50 customer records

Lab: 50 customers with FirstName, LastName, Full Name and Phone Number

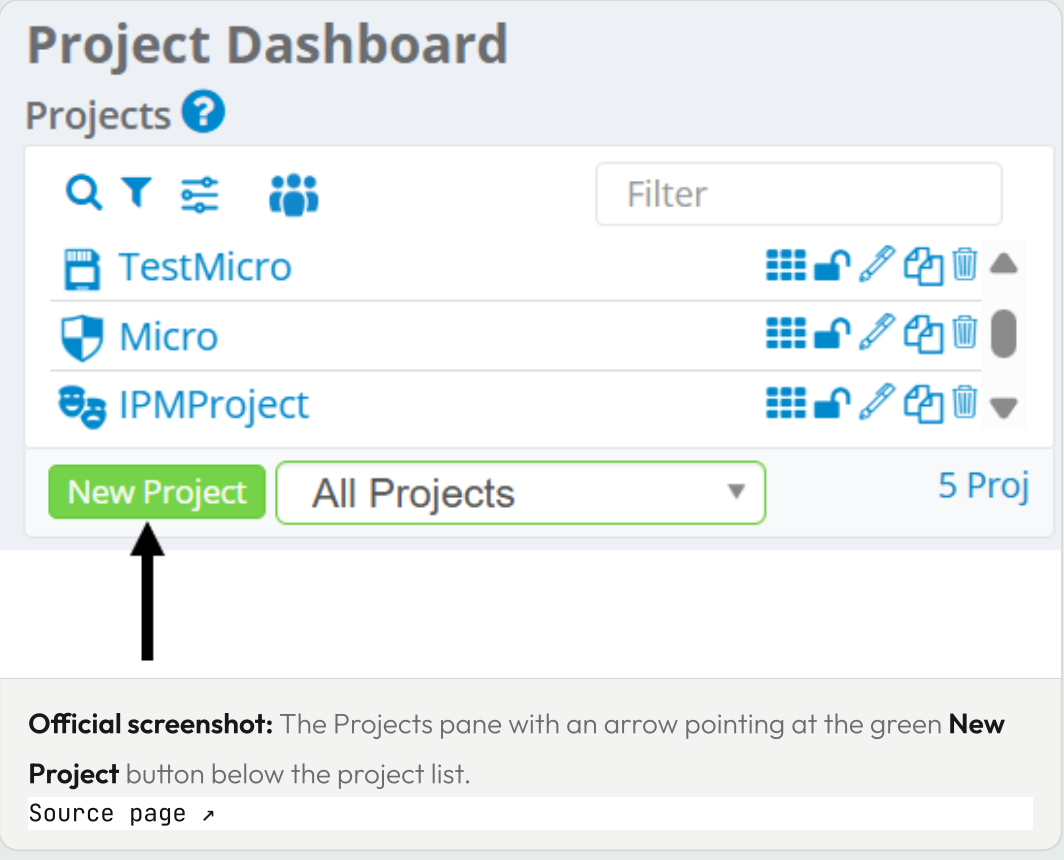
Goal: a comma-delimited file with a header row and exactly 50 data rows. Every step says what to do, why, and what you should see. Screenshots are the vendor's, from the help page named in each caption.

1

Create a project.

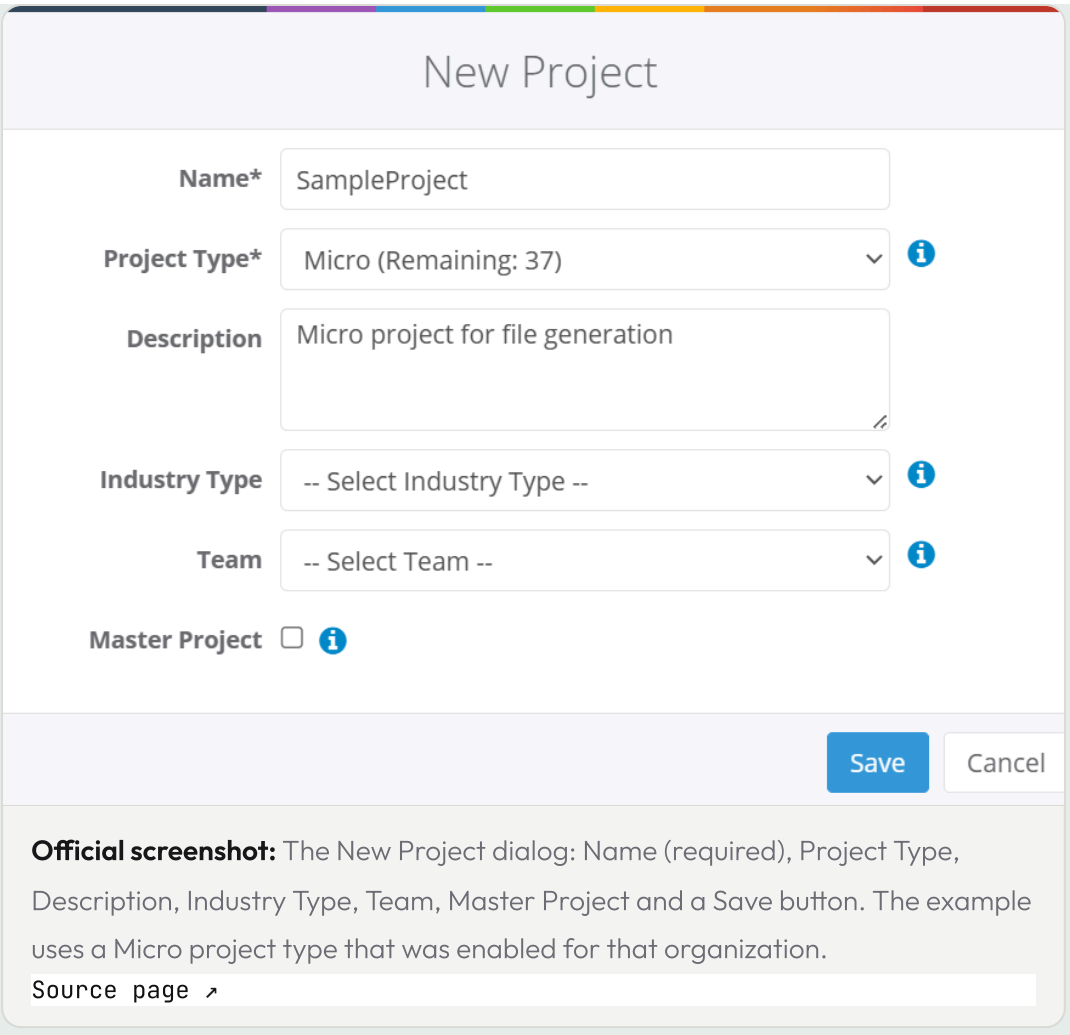
In the Projects pane of the Project Dashboard click **New Project**. Enter a name such as `ShopSphereCustomers`, add a description, and click Save. Name is the only required field; Project Type appears only if your organization has enabled it (then pick the required type). A project holds the data model for one application, and a new project automatically gets a default Project Version.

Expected result: The project appears in the Projects list, and the dashboard shows its Domains, Scenarios and other panes.



Official screenshot: The Projects pane with an arrow pointing at the green **New Project** button below the project list.

Source page ↗



Official screenshot: The New Project dialog: Name (required), Project Type, Description, Industry Type, Team, Master Project and a Save button. The example uses a Micro project type that was enabled for that organization.

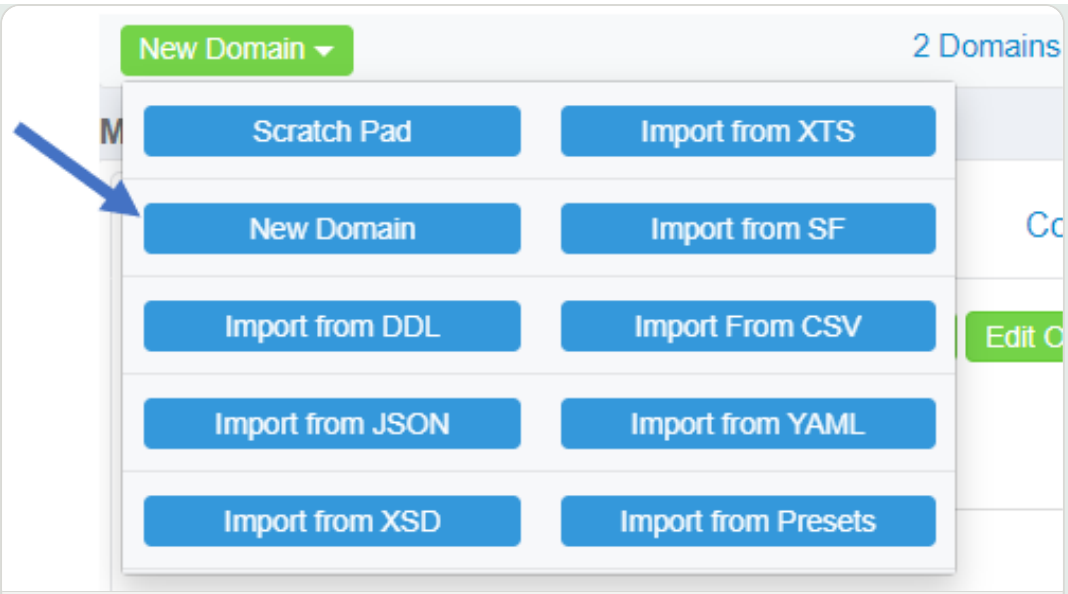
`Source page ↗`

2

Create the domain `Customer`.

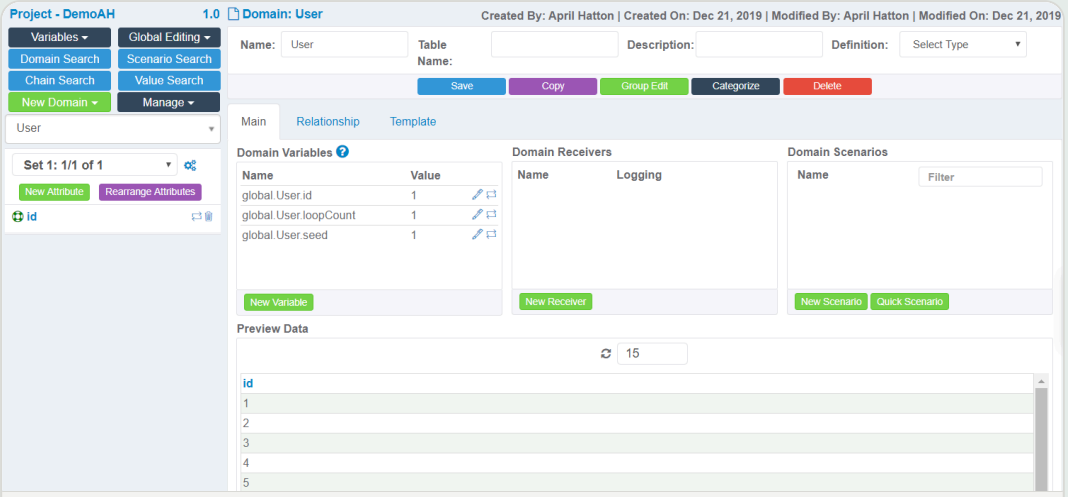
Select the project, then in the Domains pane click **New Domain** and choose **New Domain** from the menu (not Import). Enter `Customer` and Save. Domain names allow letters and numbers only. A domain is the "thing" you generate data for (see the chapter on domains).

Expected result: The Domain Dashboard opens. The domain has one attribute, `id`, and a variable `global.Customer.loopCount` equal to 1.



Official screenshot: The New Domain menu. The arrow points at the plain **New Domain** option, which creates a single domain containing only an `id` attribute. The other buttons import a data model.

Source page ↗



Official screenshot: A fresh Domain Dashboard (the example domain is called User). Look at the left attribute list, with only `id`, and the Domain Variables table with `loopCount` equal to 1 and `seed`.

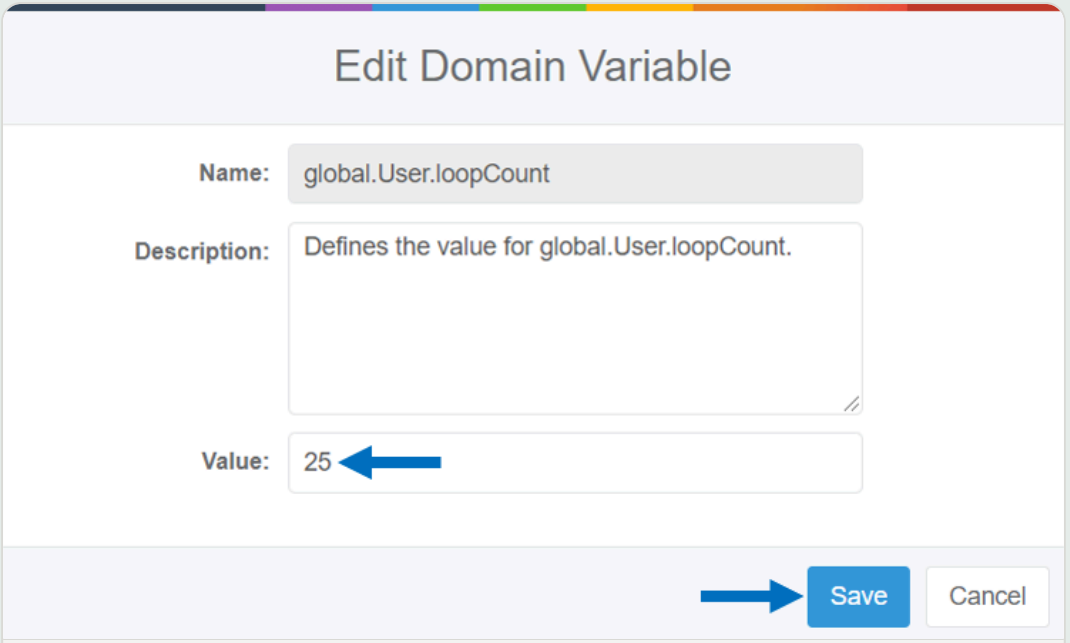
Source page ↗

3

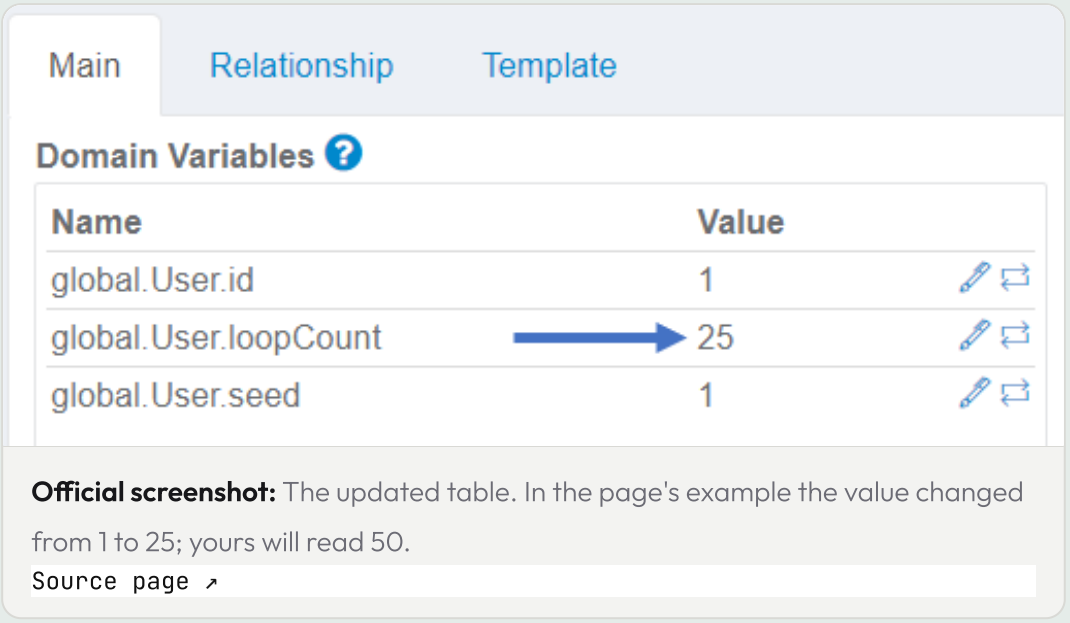
Set the loop count to 50.

In the Domain Variables table, click the pencil beside `global.Customer.loopCount`, enter 50 in Value, and click Save. The loop count is how many rows the domain generates; it defaults to 1, so skipping this gives a one-row file. Setting it on the domain (template) level cascades to scenarios.

Expected result: The Value column shows 50.



Official screenshot: The Edit Domain Variable dialog for `global.User.loopCount`. The page example types 25; you type 50.
`Source page ↗`



Official screenshot: The updated table. In the page's example the value changed from 1 to 25; yours will read 50.
`Source page ↗`

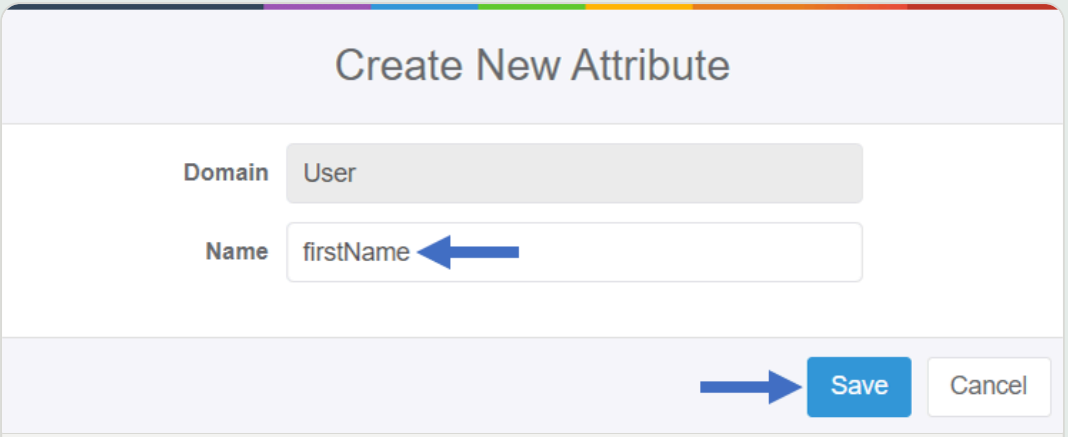
4

Add the `firstName` attribute.

In the Domain Dashboard click **New Attribute**, type `firstName` (camel case, as the platform expects), click Save. GenRocket assigns a generator by name, and for names containing "name" it chooses NameGen. Check the generator form: `nameType` firstName, `caseType` capitalize, `sortOrder` random, `waitAmount` 1.

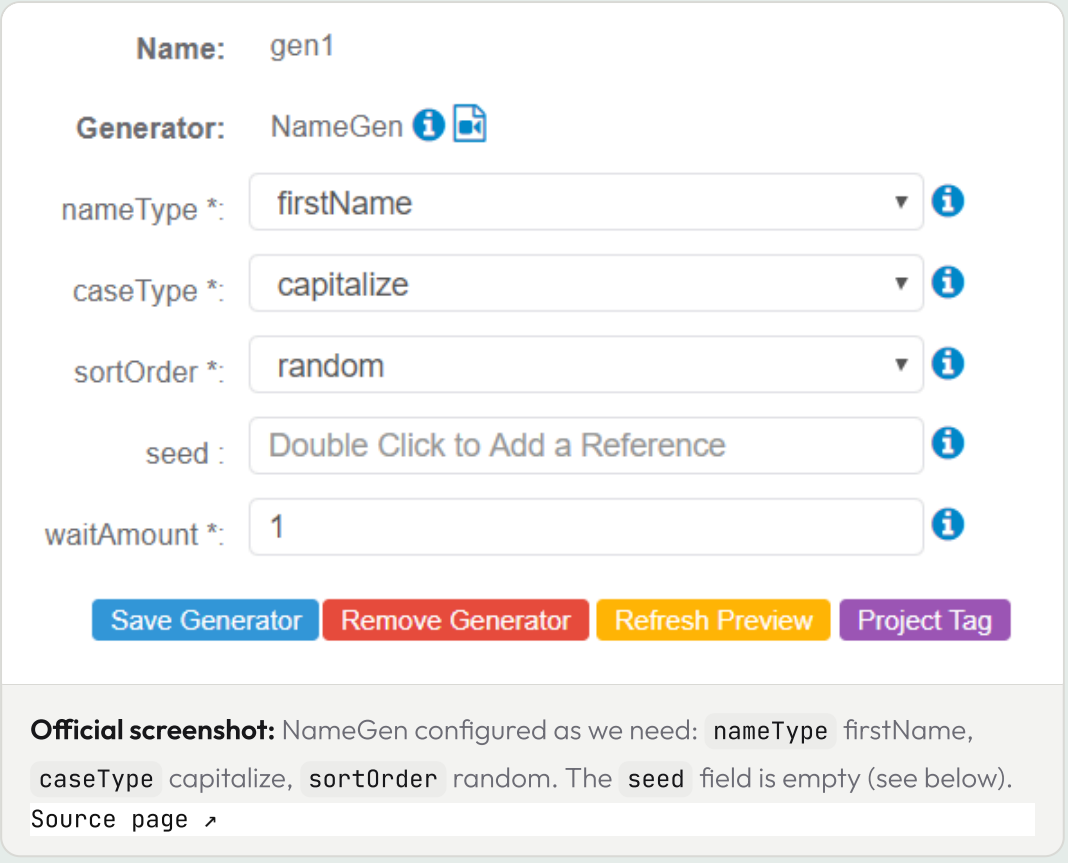
Change the fields if needed and click **Save Generator**. Capitalize gives "Asha" rather than "ASHA" or "asha"; random gives variety.

Expected result: The Preview pane lists first names such as Venice, Lonna, Ileana.



Official screenshot: The Create New Attribute dialog. Domain is fixed to the current domain and Name is typed in (arrow). Click Save.

`Source page ↗`



Official screenshot: NameGen configured as we need: `nameType` firstName, `caseType` capitalize, `sortOrder` random. The `seed` field is empty (see below).

`Source page ↗`

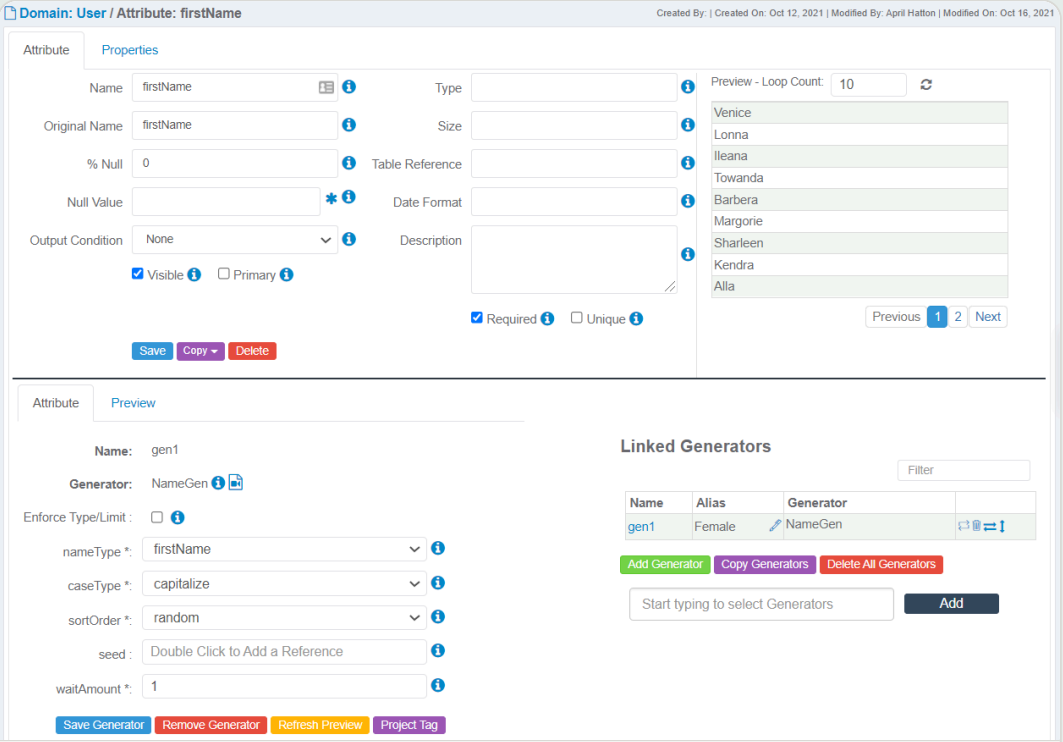
Optional: type a number such as 1 into `seed` if you want the same 50 names every run. Leave it empty for fresh names each run. This is the seeding idea from the patterns chapter.

5

Add the `middleName` attribute and hide it.

New Attribute, name `middleName`, Save. NameGen is assigned again. Keep `nameType` firstName, `caseType` capitalize and `sortOrder` random. Now, in the upper part of the Attribute Dashboard, **untick Visible** and click **Save** (the attribute Save, not Save Generator). Visible controls whether receivers see the attribute. Our file must not contain a Middle Name column, but the Full Name still needs the value.

Expected result: The attribute exists and its Visible box is unticked. The Preview still shows names.



Official screenshot: The Attribute Dashboard. Find the **Visible** checkbox below **Output Condition**. The attribute Save button is below it, and the generator form with its own Save Generator button is lower down.

`Source page ↗`

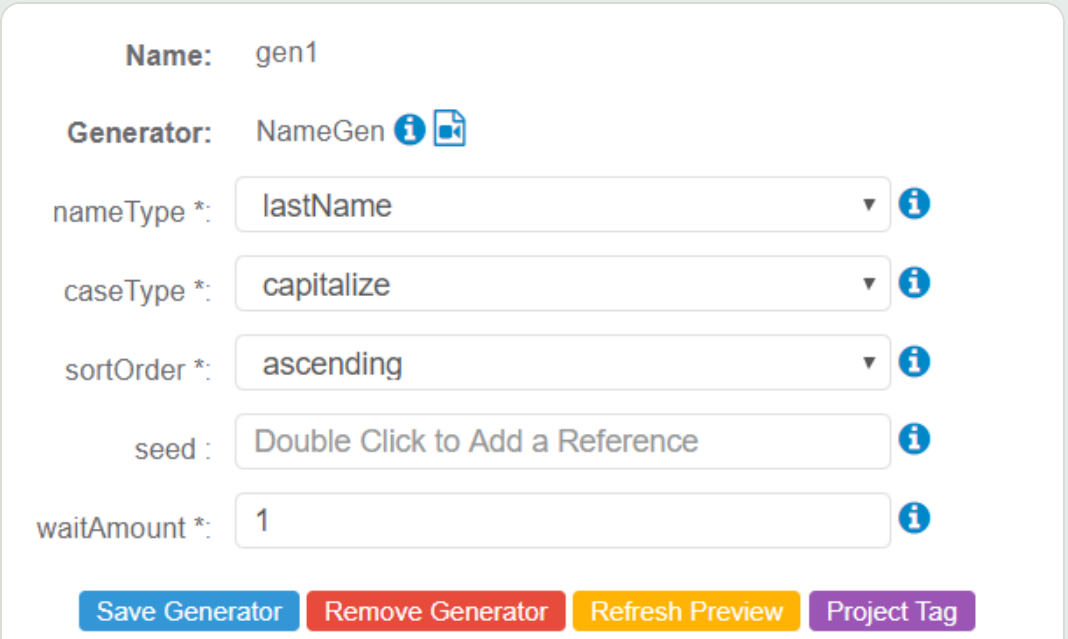
6

Add the `lastName` attribute.

New Attribute, name `lastName`, Save. In the NameGen form set `nameType` to **lastName**, keep capitalize and random, and Save Generator. Order matters: add

`lastName` after `middleName` so it sits below both, because the Full Name attribute will sit below all three and can only reference attributes above it.

Expected result: The Preview lists surnames such as Brough, Carnevale, Hollifield.



Official screenshot: NameGen with `nameType` set to lastName. This example uses `sortOrder` ascending to show alphabetical surnames; for our lab set it to random.

`Source page ↗`

7 **Add the `fullName` attribute and replace its generator with ConcatGen.**

New Attribute, name `fullName`, Save. GenRocket assigns NameGen again because the name contains "name", which is wrong here. Use the **Quick Generator Replacement** menu under the generator form: type `ConcatGen`, pick it, click **Replace**. ConcatGen joins two or more values into one string, and its `list` accepts constants and references (ConcatGen).

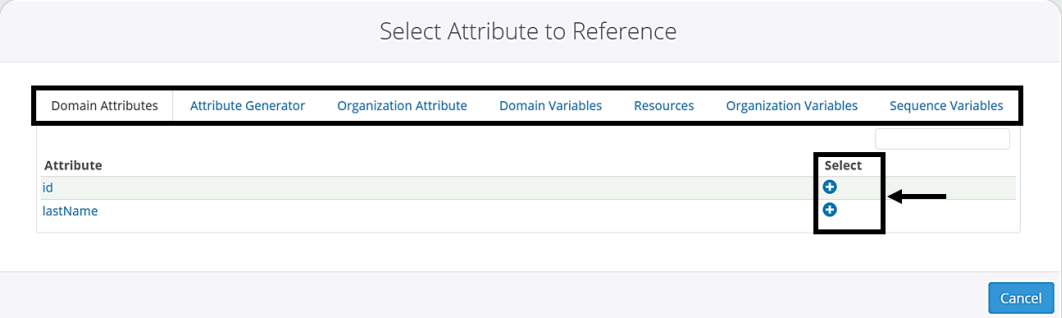
Expected result: The generator form now says ConcatGen, with an empty `list` and `caseType` noCase.

8 **Fill the `list` : firstName, a space, middleName, a space, lastName.**

Double-click the reference field under `list`. A **Select Attribute to Reference** dialog opens with tabs; on the Domain Attributes tab click the **+** beside `firstName`. Then use the **Add Space** button in the list's side menu, reference

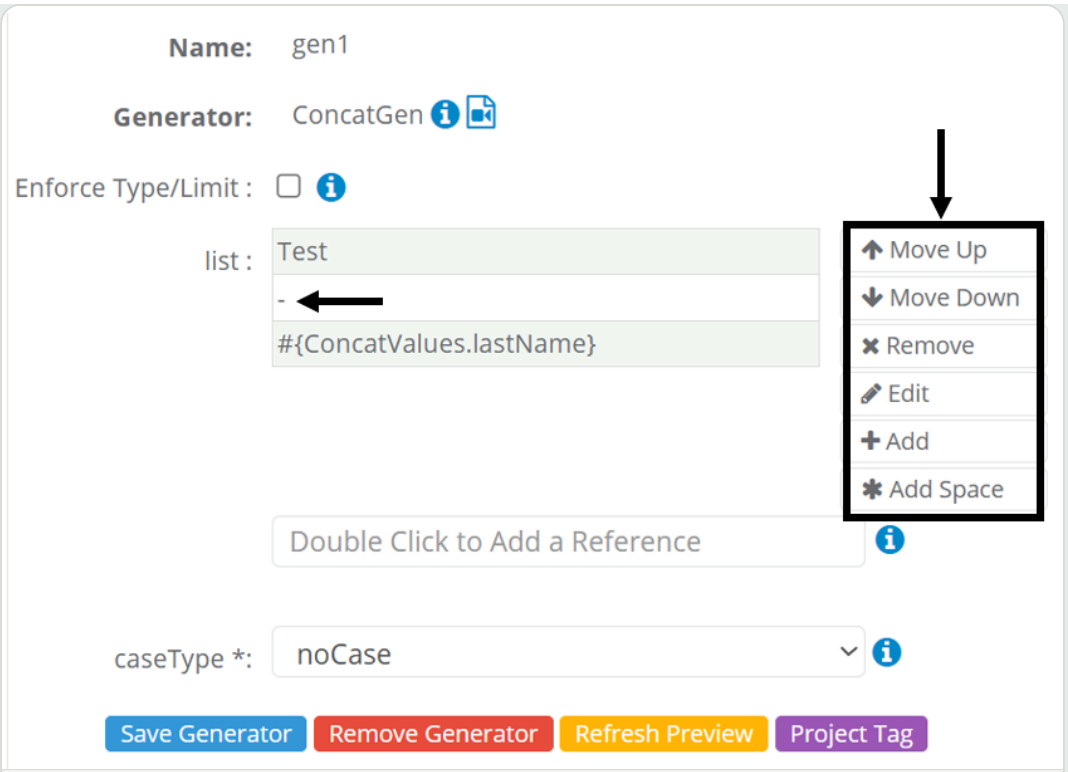
`middleName`, add another space, and reference `lastName`. Keep `caseType` noCase so capitalisation is untouched. Click **Save Generator**. The spaces are separate list items: without them you would get "AshaMenonNair". References look like `#{Customer.firstName}`, the same `#{Domain.attribute}` style the ConcatGen page shows with its example domain.

Expected result: The Preview pane shows three-word names, for example "Elba Corinne Comeaux". The first and last words match the `firstName` and `lastName` previews row for row.



Official screenshot: The **Select Attribute to Reference** dialog that opens when you double-click a reference field. Choose a tab, then click the plus beside the attribute you want (the page example picks `lastName`).

`Source page ↗`



Official screenshot: The ConcatGen `list` with a constant, a hyphen and a reference. The black box marks the side menu: **Move Up**, **Move Down**, **Remove**, **Edit**, **Add** and **Add Space**. Use it to order the items and insert spaces.

`Source page ↗`

firstName	lastName	fullName
Micheal	Buchholz	Micheal Buchholz
Lorene	Guess	Lorene Guess
Elba	Comeaux	Elba Comeaux
Elicia	Buchanan	Elicia Buchanan
Georgiana	Almaguer	Georgiana Almaguer
Angeline	Brodie	Angeline Brodie
Zelma	Sibert	Zelma Sibert
Pearlie	Dulaney	Pearlie Dulaney
Chantal	Aziz	Chantal Aziz

Official screenshot: The page's own result for a two-part name: `firstName`, `lastName` and `fullName` columns where each full name is the first name, a space and the last name. Ours adds the middle name between them.

`Source page ↗`

ORDER MATTERS

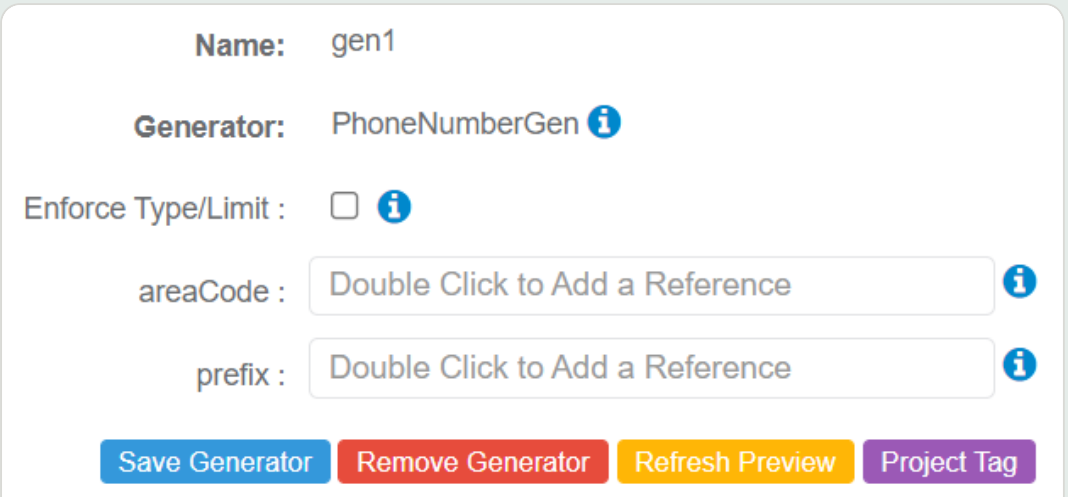
An attribute can only reference attributes *above* it. The help page "When can an Attribute not reference another Attribute?" says that referencing one below returns an empty value (link). Because we created `fullName` last, it is already below the other three. If you created attributes in another order, click **Rearrange Attributes**, drag `fullName` below `lastName`, and click Close to save.

9

Add the `phoneNumber` attribute with PhoneNumberGen.

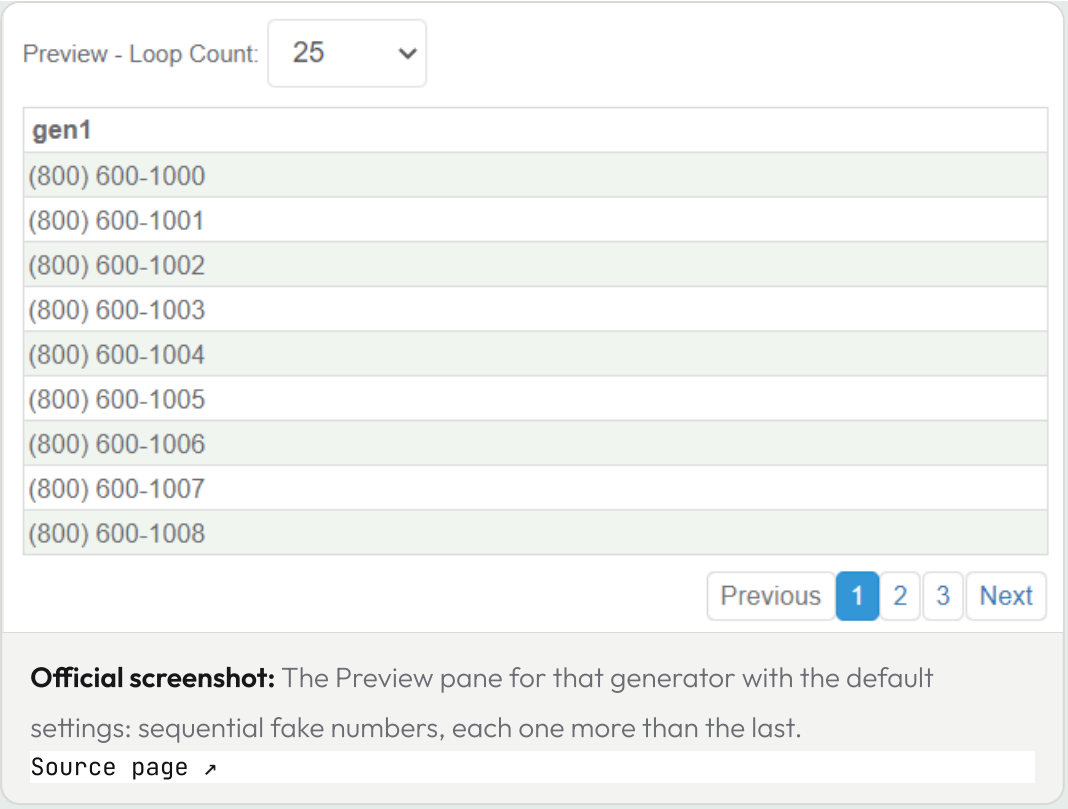
New Attribute, name `phoneNumber`, Save. If the assigned generator is not PhoneNumberGen, type `PhoneNumberGen` in the Quick Generator Replacement menu and click Replace. Its two parameters, `areaCode` and `prefix`, are optional, so leave them empty for the first run. PhoneNumberGen generates numbers in sequence, so every row gets a different phone number, which suits a column that tests often treat as unique.

Expected result: The Preview shows consecutive numbers. In the vendor's screenshot they look like (800) 600-1000, (800) 600-1001 and so on; your area code and format may differ.



Official screenshot: The PhoneNumberGen form: two optional fields, `areaCode` and `prefix`.

`Source page ↗`



Official screenshot: The Preview pane for that generator with the default settings: sequential fake numbers, each one more than the last.

`Source page ↗`

10 **Review all five attributes together.**

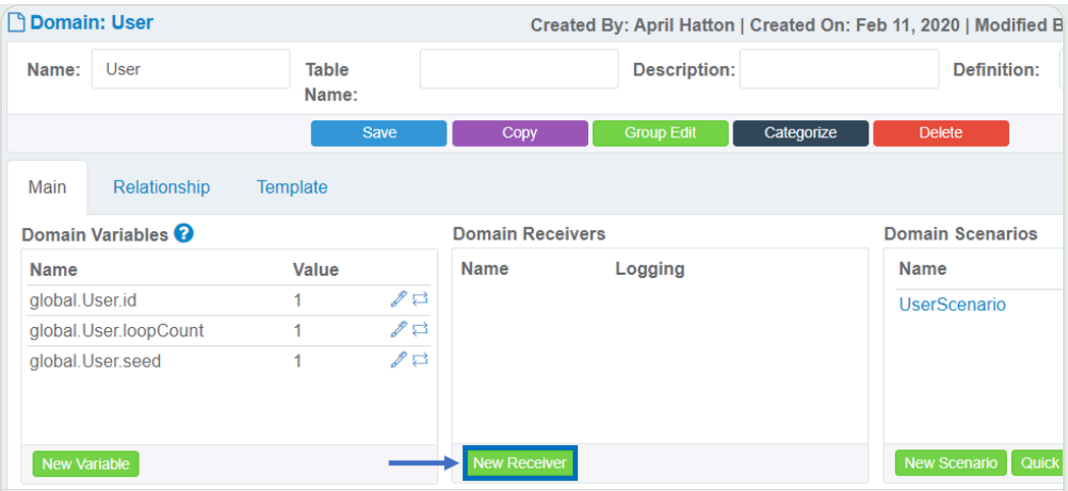
Back on the Domain Dashboard look at the attribute list (id, firstName, middleName, lastName, fullName, phoneNumber, in that order). Check the preview of each. This is the cheapest place to find a mistake. A wrong generator costs seconds here and a re-download later.

Expected result: Six entries in the list, and the Preview Data area on the dashboard shows rows. The attribute `middleName` appears in the preview even though it will not appear in the file.

11 **Add the receiver.**

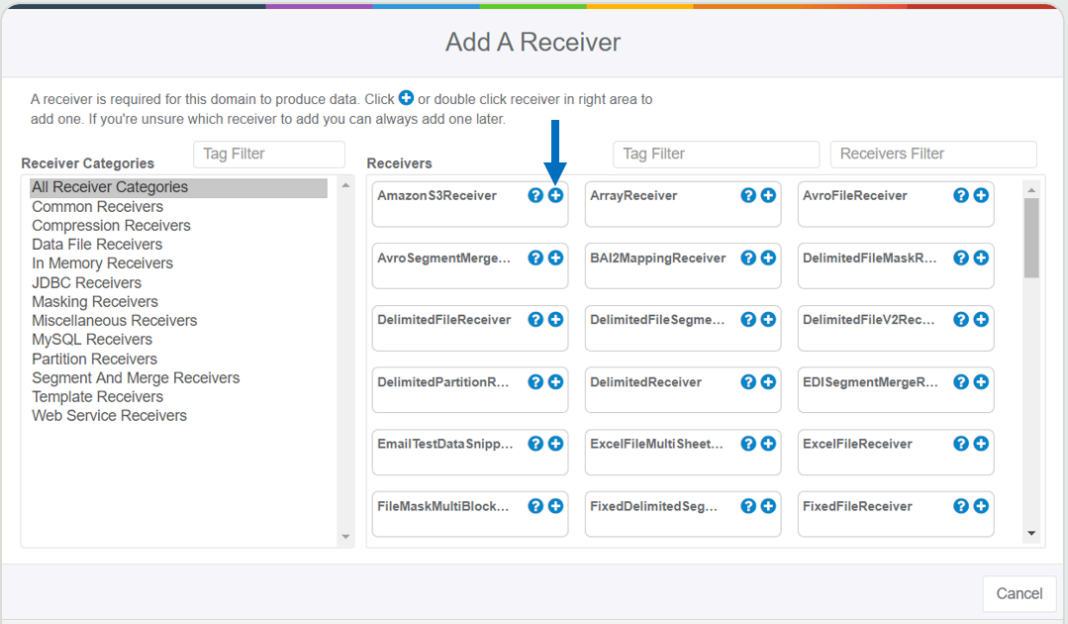
In the Domain Receivers pane click **New Receiver**. In the Add A Receiver dialog use the Filter box and click the **+** on **DelimitedFileReceiver**. Accept or edit the receiver name, click Save. A receiver "receives unformatted data (raw data) from a GenRocket Generator" and turns it into an output format (Receivers Overview). Add the receiver *before* creating the scenario: if a scenario already exists, the dialog asks which scenarios should get the receiver, and a scenario with no receiver produces no output.

Expected result: The receiver is listed in the Domain Receivers pane and its parameter form opens.



Official screenshot: The Domain Dashboard of a `User` domain with an arrow at the green **New Receiver** button in the Domain Receivers pane.

`Source page ↗`



Official screenshot: The **Add A Receiver** dialog: categories on the left, receiver tiles on the right (AmazonS3Receiver, ArrayReceiver, DelimitedFileReceiver and more). Click the plus on the tile you want.

`Source page ↗`

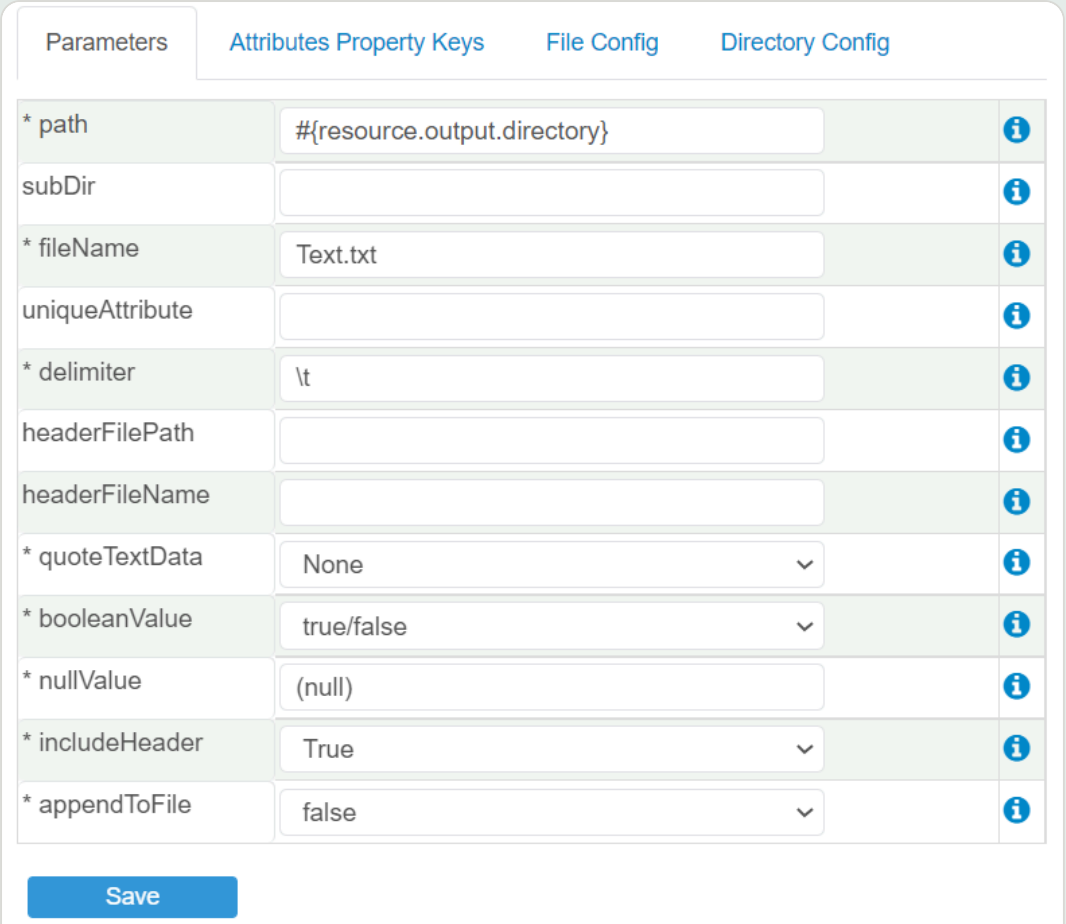
12

Fill in the receiver parameters.

Required parameters have an asterisk. For the DelimitedFileReceiver they are `path` (where the file goes), `fileName`, `delimiter`, `booleanValue`, `nullValue`, `includeHeader` and `appendToFile` (DelimitedFileReceiver). Set them like this, then click **Save**:

- `path` : leave the pre-filled `#{resource.output.directory}` . It points at your personal output folder, so the scenario works on any user's machine.
- `fileName` : `Customers.csv` .
- `delimiter` : `,` (the screenshot's default is a tab, shown as `\t`).
- `includeHeader` : True, so the first line holds the column names.
- `appendToFile` : false, so each run replaces the file instead of adding 50 more rows to the old ones.
- `quoteTextData` : None. Leave `booleanValue` and `nullValue` at their defaults for now.

Expected result: The receiver form shows your values and the Save succeeds. The receiver's tabs include Parameters and Attributes Property Keys.



Official screenshot: The DelimitedFileReceiver parameter form with its defaults: `path` `#{resource.output.directory}` , a sample `fileName` , tab `delimiter` , `includeHeader` True and `appendToFile` false. You edit `fileName` and `delimiter` .

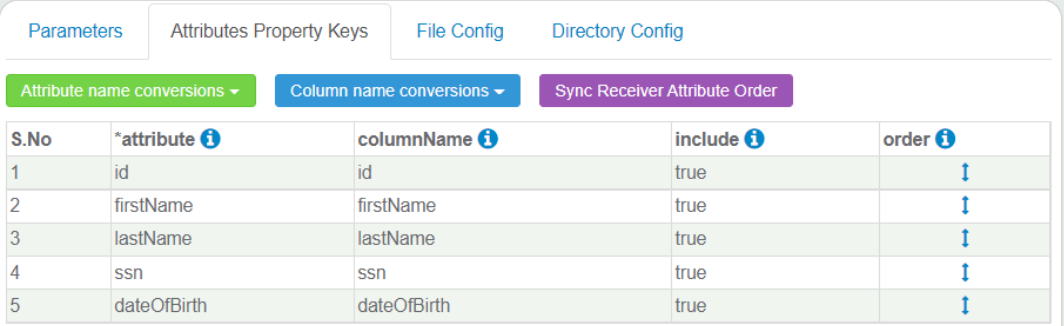
Source page ↗

13

Set the column headers and drop the `id` column.

Open the receiver's **Attributes Property Keys** tab. The receiver defines two property keys per attribute: `columnName` ("the column name as it will be output in the delimited file header") and `include` ("if the Attribute will be included as a column in the output"). Without a header file, the header is each attribute's original name. Click each attribute and set `columnName` to `FirstName`, `LastName`, `Full Name` and `Phone Number`, set `include` to false for `id` (and for `middleName` if it is listed), and save. The Global Edit button changes several at once. By default the file's column order follows the attribute order; the tab also lets you drag attributes to reorder.

Expected result: The table lists attributes with their column names and include flags, with only the four wanted columns marked true.



Official screenshot: The **Attributes Property Keys** tab: one row per attribute with `attribute`, `columnName`, `include` and an `order` handle. The page's example domain has id, firstName, lastName, ssn and dateOfBirth.

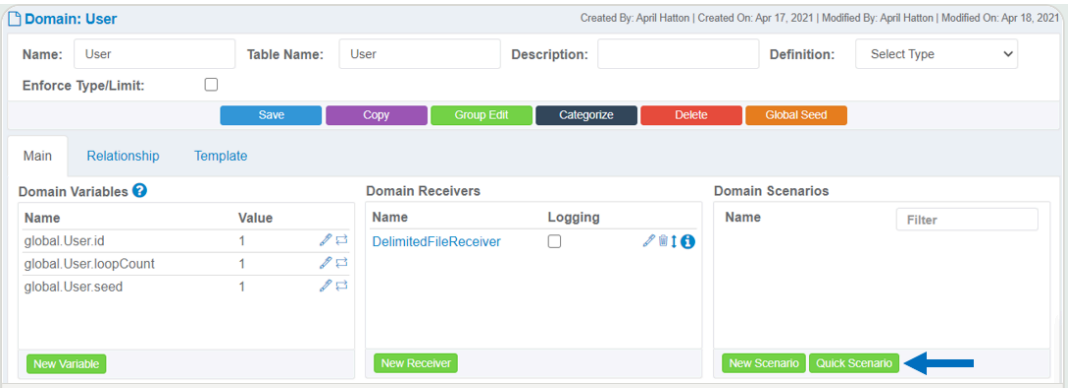
`Source page ↗`

14

Create the scenario.

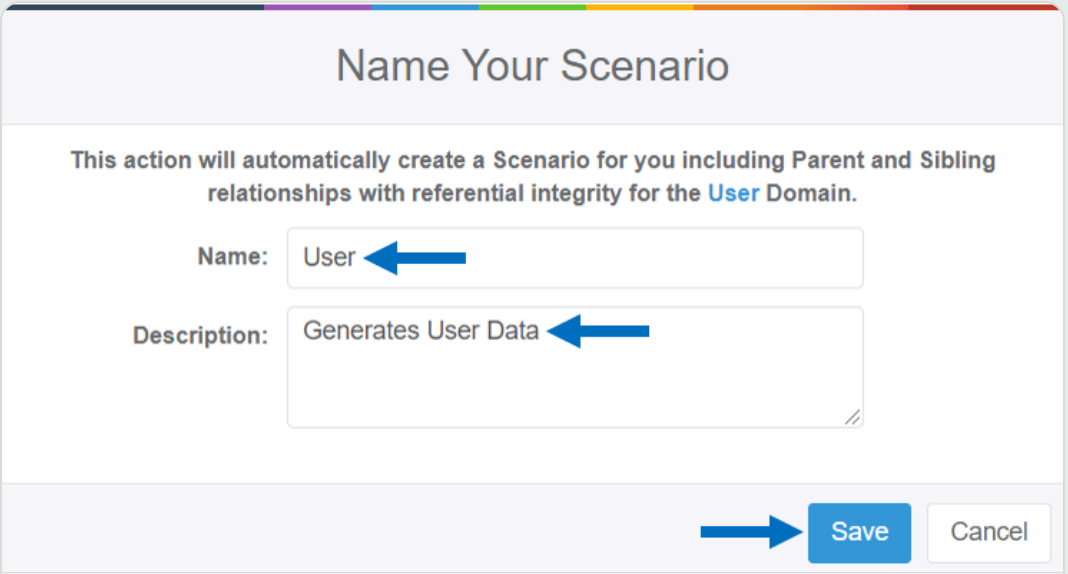
A scenario "contains instructions that tell the GenRocket engine what synthetic data to generate and how much" (Scenarios Overview). The quickest route is on the Domain Dashboard: click **Quick Scenario**, keep the suggested name (it matches the domain), add an optional description, and Save. Quick Scenario also adds any parent and sibling domains needed for referential integrity; we have none. (How do I create a Scenario? also describes New Scenario, where you choose the primary domain yourself.)

Expected result: You land on the Scenario Dashboard. Its Scenario Domains table lists `Customer` with Loop Count Value `#{global.Customer.loopCount}`, which resolves to 50, and no "does not have any Receivers" notice.



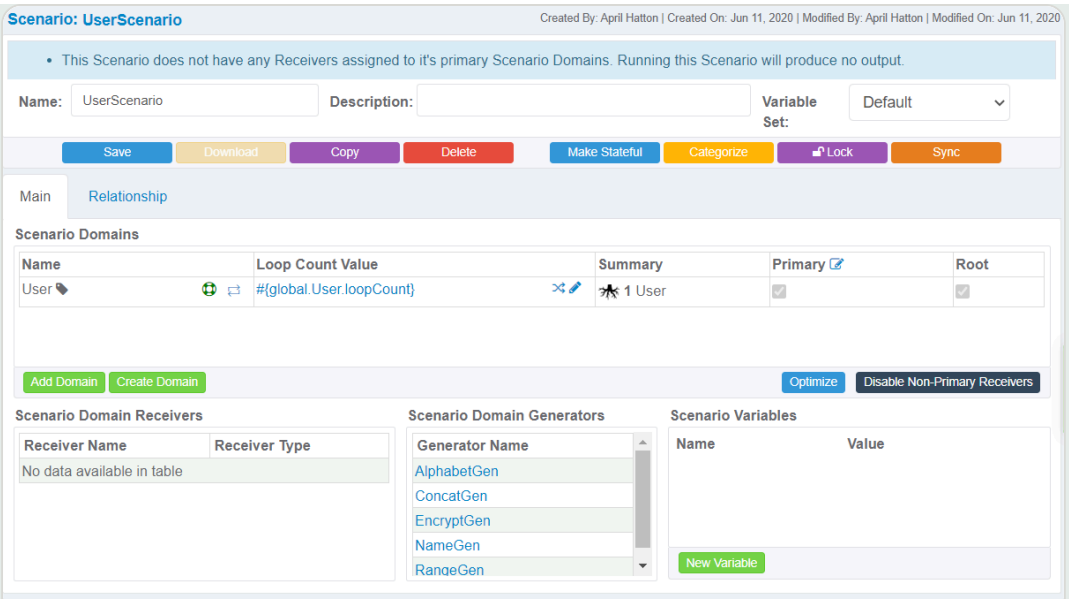
Official screenshot: The Domain Dashboard with an arrow at the **Quick Scenario** button next to New Scenario in the Domain Scenarios pane.

`Source page ↗`



Official screenshot: The Quick Scenario dialog: name (pre-filled with the domain name), optional description and Save.

`Source page ↗`



Official screenshot: A Scenario Dashboard. The blue notice says this scenario has no receivers, which is why we added the receiver first. The Scenario Domains table shows the loop count reference and the Download button is in the button row above it.

`Source page ↗`

15

Download the scenario.

Click the **Download** button on the Scenario Dashboard. The scenario runs on your own machine, so the platform gives you a `.grs` file. This is the design half finished. If your browser warns about the file type, keep it, as the user-profile page notes for downloaded files.

Expected result: A file named after the scenario (for example `Customer.grs`) in your downloads folder.

Official screenshot: The orange **Download** button of the Scenario Dashboard, which saves the scenario file to your computer.

`Source page ↗`

16

Run the scenario.

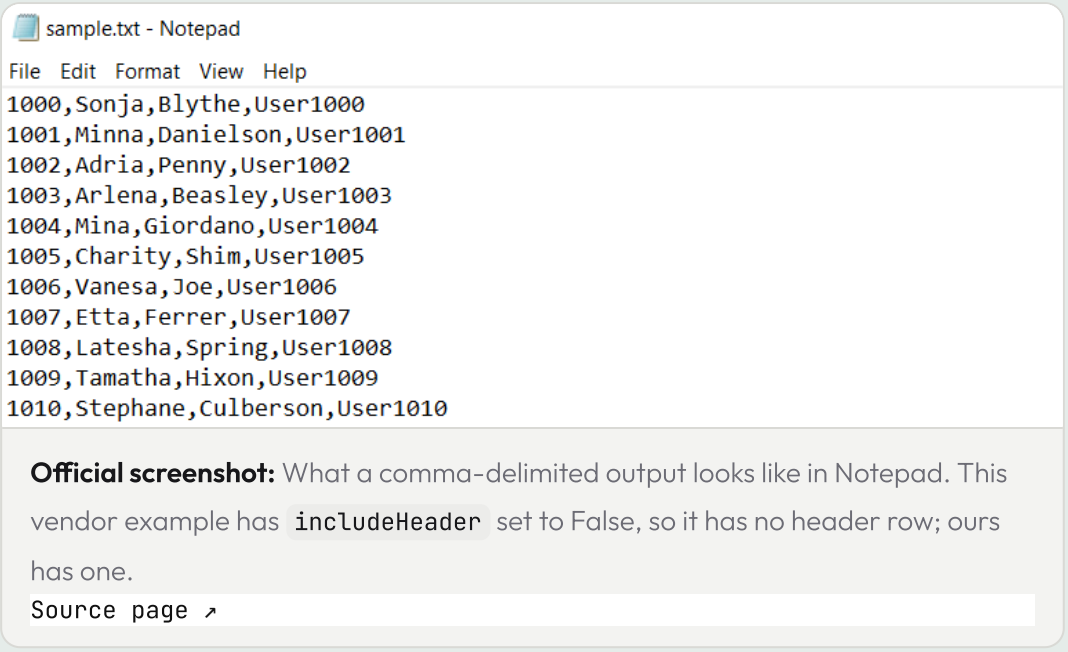
Open a command window or terminal and go to the folder holding the downloaded file. Run `genrocket -r Customer.grs`, replacing the name with yours. The run page says a scenario must be run in a command window or terminal on your local computer, and that success is silent: if data loads correctly there is no exception, and a problem shows up as an exception on the command line (How do I run a GenRocket Scenario?).

Expected result: The command returns without an error message. A file `Customers.csv` appears in the folder you set for `resource.output.directory`.

17 **Open the file.**

Open `Customers.csv` in a text editor or spreadsheet. Then do the checks in the next section before calling the lab done.

Expected result: A header line plus 50 data lines.



Official screenshot: What a comma-delimited output looks like in Notepad. This vendor example has `includeHeader` set to False, so it has no header row; ours has one.

Source page ↗

7.18.1 How the vendor's own demo compares with this lab

In the 2020 webinar, GenRocket's CTO builds and runs a user data set in about ten minutes of screen time. Setting his path beside the lab shows which steps are essential and which are choices. The demo is a vendor presentation of an older version, so follow the lab and the docs for clicks.

The 2020 demo path and this lab

Step	2020 demo	This lab
Domain	Made with Scratch Pad: attribute names typed in a list, generators assigned automatically	New Domain, then one attribute at a time (steps 2 and 4 to 9), so you see each generator
Rows	The scenario generates 10,000 users	50, set in step 3. The docs say a new domain starts at 1, so always set it
Receiver	A flat XML file receiver, mostly default settings	A DelimitedFileReceiver writing CSV (steps 11 to 13)
Scenario	Download button; he shows a small encrypted instruction file, about 22 KB, and says the size is the same for one row or a billion	Create, then download (steps 14 and 15)
Run	The Java-based runtime from a command line	`genrocket -r` (step 16)
Check	Scrolls the output: the id sequence, names, SSNs, card numbers and the hash, then the bottom of the file to confirm 10,000 rows	The checks in section 7.20

The flow is the same: model, choose a receiver, build a scenario, run it locally, read the file. His last move, counting the rows at the bottom, is the cheapest check there is, so section 7.20 adds it, with a spot-check of the sequence.

7.19 What the output should look like

Your file will contain different names, but the shape must match. The sample below is example data (not captured from a real run); the phone format follows the vendor's PhoneNumberGen preview screenshot.

FIRST 5 DATA ROWS OF CUSTOMERS.CSV (PLUS THE HEADER)

FirstName	LastName	Full Name	Phone Number
Venice	Brough	Venice Alecia Brough	(800) 600-1000
Lonna	Carnevale	Lonna Hector Carnevale	(800) 600-1001
Ileana	Hollifield	Ileana Rosalind Hollifield	(800) 600-1002
Towanda	Studer	Towanda Elwood Studer	(800) 600-1003
Barbera	Ignacio	Barbera Kuehl Ignacio	(800) 600-1004

In the raw file each row is one comma-separated line, for example `Venice,Brough,Venice Alecia Brough,(800) 600-1000`. Notice that the Full Name's first word equals FirstName and its last word equals LastName. That relationship is the proof that the reference approach worked.

7.20 Verification checks

A file that "looks fine" in the first few rows can still be wrong in row 37. Check it deliberately. The commands below are general shell commands (macOS, Linux or Git Bash on Windows), not from the GenRocket documentation; a spreadsheet can do the same with COUNTA, LEN and conditional formatting.

Checks, how to run them and what you expect

Check	Command	Expected
Row count	`wc -l Customers.csv`	51 (header plus 50). It shows 50 if the file has no final newline, so also check the last row number.
Header names	`head -1 Customers.csv`	`FirstName,LastName,Full Name,Phone Number`
Four columns per row	`awk -F, 'NR>1 && NF≠4' Customers.csv`	No output.
No empty values	`awk -F, 'NR>1{for(i=1;i⩽NF;i++) if($i=="") bad++} END{print bad+0}' Customers.csv`	`0`
Full Name consistent	`awk -F, 'NR>1{n=split($3,p," "); if(n≠3||p[1]≠$1||p[3]≠$2) print NR}' Customers.csv`	No output.
Phone length	`awk -F, 'NR>1{print length($4)}' Customers.csv | sort | uniq -c`	One line: 50 rows with the same length (14 characters for the `(800) 600-1000` format).
Phones unique	`cut -d, -f4 Customers.csv | tail -n +2 | sort | uniq -d`	No output (no duplicates).
Last row (the demo's "scroll to the bottom")	`tail -n 2 Customers.csv`	A complete row, not cut off, in the same four-column shape as the first row.
Sequence spot-check (also from the demo)	`sed -n '2p;51p' Customers.csv`	The two phones differ by 49 if your sequence steps by one: the 50th is the 1st plus 49.

Full checklist before you call the lab finished:

- The file is where `resource.output.directory` says, and is named `Customers.csv`.
- Exactly 50 data rows; the header is present once.
- Only the four wanted columns: no `id`, no middle name.
- No empty or "null" cells in any column.
- Full Name equals FirstName, a middle name and LastName for every row.
- All phone numbers have the same length and are unique.
- Running the scenario a second time still gives 50 rows, not 100 (`appendToFile` false). If you set a `seed`, the names are the same as before.

> **WHY VERIFY AT ALL?**
>
> Test data is an input to other tests. If the data silently has 49 rows or an empty phone, every test that uses it can fail for reasons that have nothing to do with the application. A five-minute check protects hours of debugging.

7.21 Troubleshooting

Problems, causes and fixes (documentation-based unless marked)

Symptom	Likely cause and fix
No New Project button	You lack permission to create projects; ask the organization admin (project creation page).
`'genrocket' is not recognized` or `command not found: genrocket`	The Runtime folder or environment variables are wrong. Check the `genrocket` folder is in your home directory, `GEN_ROCKET_HOME` is set and `PATH` includes it (Runtime setup warnings and errors).
`Could not find the required profile: GRXXXXXXXProfile.grp`, "INVALID ACCESS KEY"	Download your user profile and place it in the `.genrocket` folder, replacing an older one, then re-run (error page).
"Unable to Download Users Profile"	Every resource in the dashboard needs a valid value; enter N/A for resources you do not use.
Run succeeds but there is no file	The receiver is missing or disabled for the scenario, or the output folder is not where you looked. Check the Scenario Dashboard's receiver pane and your `resource.output.directory`.
Only 1 row	The domain loop count is still 1. Set `global.Customer.loopCount` to 50, and download the scenario again.
Changed something online, nothing changed in the file	The scenario file is a snapshot. Download it again after every change. The DelimitedFileReceiver page says this explicitly for `includeHeader`, and adds that an outdated receiver jar can still show headers.
Full Name column empty	`fullName` is above the attributes it references. Use Rearrange Attributes.
Full Name has no spaces	Add Space items missing from the ConcatGen `list`.

Symptom	Likely cause and fix
A MiddleName column appears	`Visible` still ticked, or `include` true on the receiver. Untick it and re-download.
Same names every run	A `seed` is set on NameGen. Clear it for fresh names.
`OutOfMemoryError: GC overhead limit exceeded`	Unlikely at 50 rows; for large runs set `JAVA_OPTS` and avoid tight uniqueness limits (error page).

7.22 Extension lab: an email and a nullable column

Lab: add Email and a nullable Alternate Phone

Goal: the same 50 rows with two more columns, one of them sometimes empty. This puts the null pattern from the null and blank section of the patterns chapter into practice.

1. **Add `email` with EmailGen.**
   New Attribute `email`; names containing "email" get EmailGen automatically. Its parameters are `prefix` (the local part), `reference` (another attribute, to make the local part unique), `suffix` (the first part of the domain) and `extension` (com, net and so on) (EmailGen). Set `prefix` `user`, `reference` the `id` attribute, `suffix` `example`, `extension` `com`. Referencing `id` makes each address unique.

   Expected result: Preview shows addresses like `user1@example.com` (based on the pattern in the vendor's API example).

2. **Add `alternatePhone` and make it nullable.**
   Give it PhoneNumberGen with `areaCode` 900 so it differs from the main phone. On the Attribute Dashboard set **% Null** to 30. That field is "the percentage of time the null value is used for negative testing". Click Save.

   Expected result: Preview shows a mix of numbers and empty or null cells.

3. **Decide how null is written.**
   On the receiver, check `nullValue` (default `(null)` in the screenshot). Set it to an empty string if your consumer expects an empty cell. Add the two attributes to the

receiver's Attributes Property Keys with column names `Email` and `Alternate Phone`.

4 **Download and run again.**
The earlier checks change: six columns; no empty values in the first four; roughly 15 of 50 `Alternate Phone` cells empty (the docs give no exact count for `% Null`).

▶ **Idea: ten-digit phone numbers instead**

7.23 Extension lab: customers with a varying number of orders

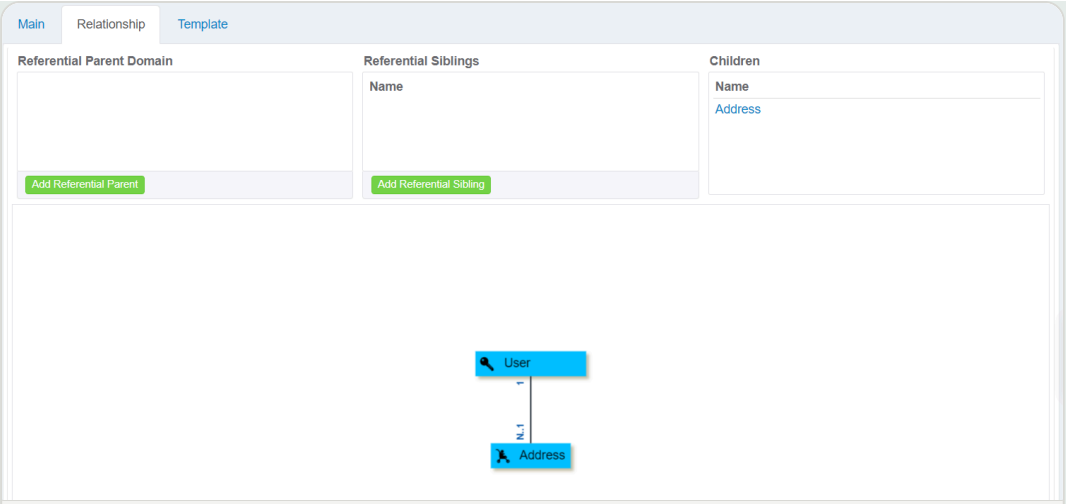
Real data has parents and children: one customer, several orders. GenRocket models this with a parent domain and a child domain, and each domain "has one parent but many siblings" (manual parent and child relationships). This lab follows the vendor's page "How to generate Children based on Weights (Percentages)" (link), with ShopSphere names.

Lab: 10 customers, each with 1, 2 or 3 orders

Goal: 10 customers; 5 of them (50 percent) have 2 orders, 2 (20 percent) have 3 and 3 (30 percent) have 1. Total orders: 19.

1 **Create the child domain and set the parent.**
Create a domain `Order` with a RangeGen order id and a delimited receiver (as in the main lab). In `Order`'s Domain Dashboard open the **Relationship** tab, click **Add Referential Parent**, choose `Customer`, Save. Set `Customer`'s loop count to 10. A parent domain must exist so that each order points at a real customer.

Expected result: A small diagram shows Customer above Order.



Official screenshot: The Relationship tab drawing of the vendor example: a `User` box above an `Address` box. Ours is Customer above Order.

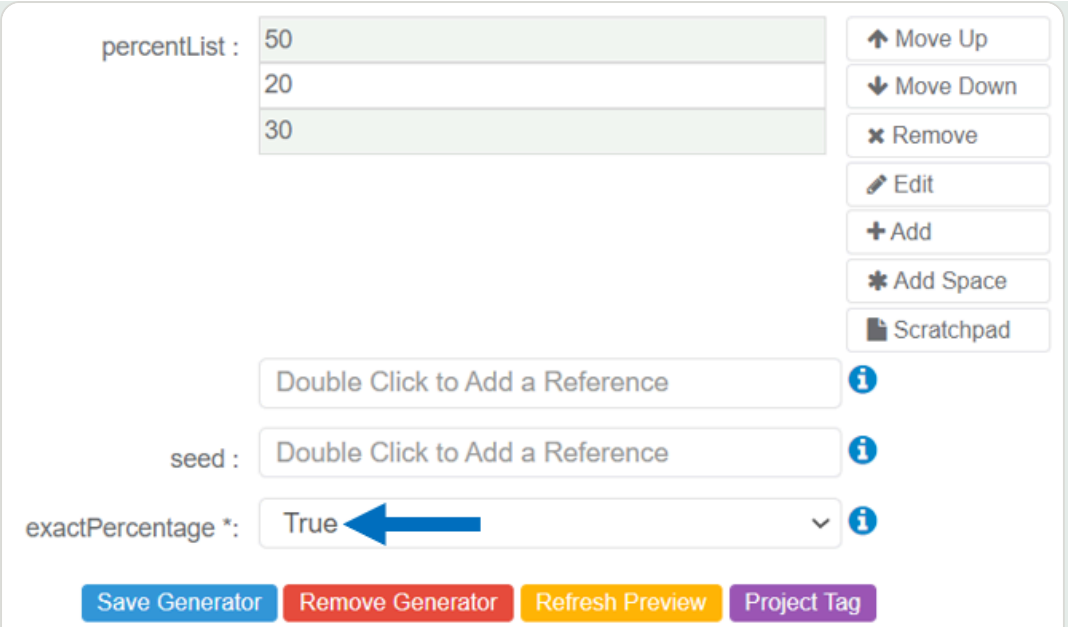
`Source page ↗`

2

Add an attribute to the parent that holds the child count.

In `Customer`, add `childLoopCount`, untick Visible (it is a helper), and replace its generator with **MultiWeightGen**: `valueList` 2, 3, 1; `percentList` 50, 20, 30; `exactPercentage` True. Save Generator.

Expected result: The attribute exists, hidden, and its preview shows 2, 3 and 1.



Official screenshot: The percent list (50, 20, 30) and `exactPercentage` True on the vendor's example.

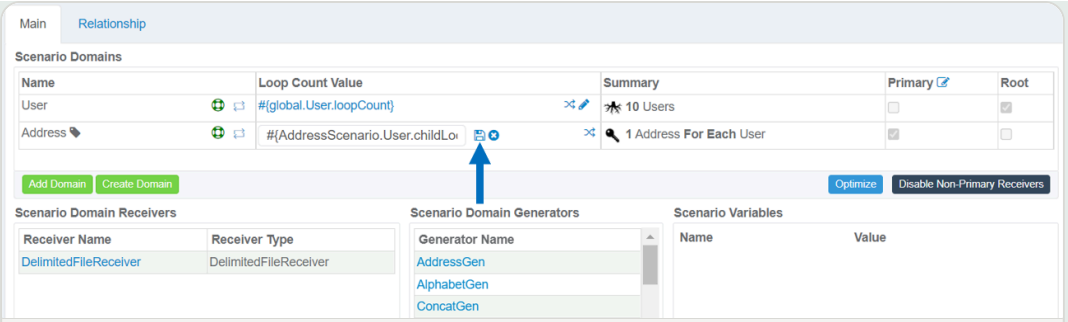
`Source page ↗`

3

Create the scenario for the child and point its loop count at the parent's attribute.

Create a scenario with `Order` as the primary domain (Quick Scenario adds the parent). On the Scenario Dashboard double-click `Order` 's Loop Count Value, choose the Domain Attributes tab, select `Customer` , click **+** beside `childLoopCount` , and save with the disk icon.

Expected result: The Summary column reads "N Order for each Customer", where N comes from `childLoopCount` .



Official screenshot: The Scenario Domains table while editing: the child's Loop Count Value field holds a reference to the parent's `childLoopCount` (the disk icon saves it). The Summary still reads "1 Address For Each User" until saved; the page says it then reads "N Address For Each User".

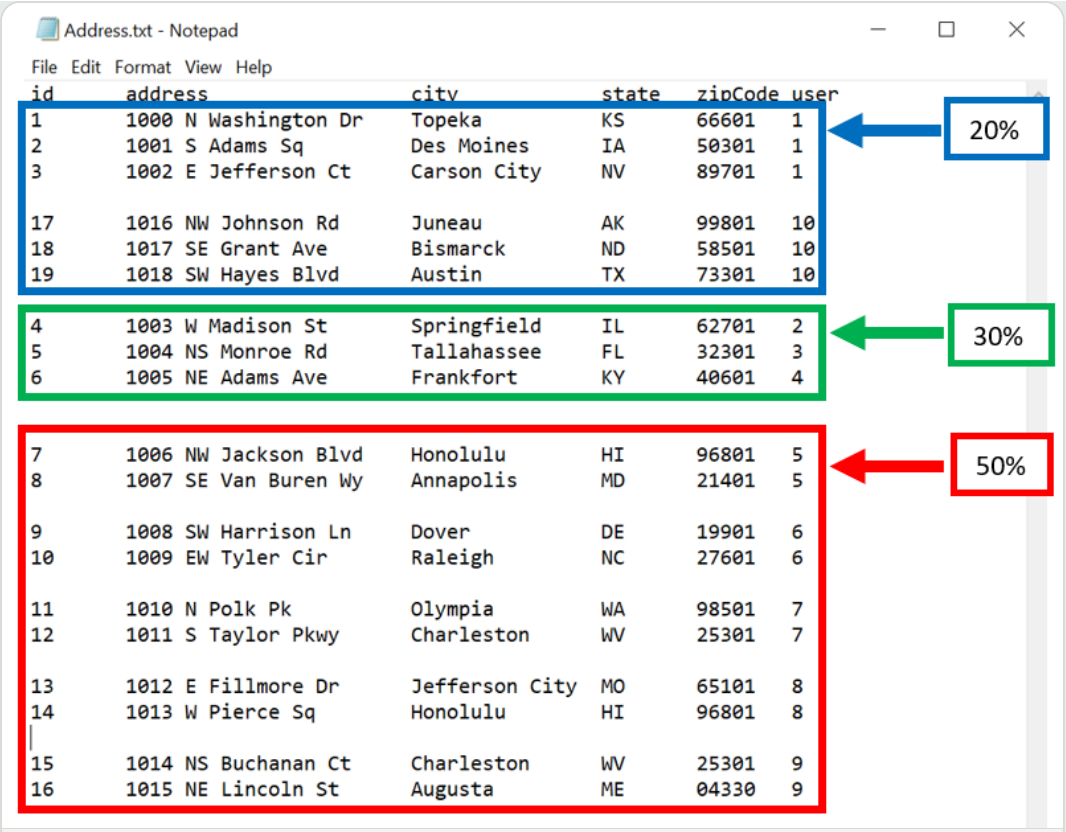
`Source page ↗`

4

Download and run.

Run `genrocket -r <scenario>.grs` . The vendor's output shows ten users with 3, 1 or 2 addresses in the stated proportions, as the annotated screenshot shows.

Expected result: 19 order rows. Count orders per customer id: five customers with 2, two with 3, three with 1.



Official screenshot: The vendor's output with the groups outlined: 20 percent of users have 3 addresses, 30 percent have 1 and 50 percent have 2.

`Source page ↗`

▶ **Show a foreign key column and notes**

7.23.1 Third lab: the boundary suite

The third lab is in the patterns chapter: a boundary-value suite for ShopSphere orders. It reuses steps 11 to 16 here (receiver, scenario, download, run) with a ListGen whose `setLoop` is True, so the row count is the number of boundary cases, not 50. Run it, then check that you get exactly seven rows.

7.24 Recap

RECAP

- A TDG plan has four stages: use cases, proof of concept, environment assessment and deployment plan. The 50-record lab is a POC.
- Lab path: project, domain `Customer`, loop count 50, five attributes, a DelimitedFileReceiver, a scenario, download, `genrocket -r`, open the file.
- FirstName and LastName use NameGen (`nameType` firstName and lastName); the middle name uses NameGen too, because no middle-name type is documented.
- Full Name is a ConcatGen whose `list` references the three name attributes with spaces between; it must sit below them, and the helper attribute is hidden with Visible off.
- Phone Number is PhoneNumberGen, a sequence of numbers.
- The receiver's `columnName` and `include` property keys set the headers and drop `id`; `appendToFile` false keeps reruns at 50 rows.
- Verify the output: 50 rows, four columns, no empties, consistent Full Name, equal-length unique phones. Download the scenario again after every change.
- Extensions: EmailGen and `% Null` for a nullable column; MultiWeightGen for child counts; ListGen with `setLoop` for boundary rows.