EVL Data Generation

version 2.8

team@evltool.com

https://www.evltool.com/evl-data-generation

This manual is for **EVL Data Generation** (version 2.8), an EVL Microservice which provides an automated but flexible way to generate data. Just based on a configuration file all data generation jobs are generated.

Table of Contents

1 Introduction

EVL Anonymization Microservice enables fast, automated and cost-effective anonymization of data sets. It can be used for anonymization of the production data according to GDPR requirements as well as for the protection of commercially sensitive data for developers, testers and other outside contractors.

EVL Anonymization belongs to the portfolio of other **Metadata-driven EVL Microservices**, which provide fast and automated solution for a specific business purpose, but keep common metadata structure so other such EVL Microservices can be plugged-in easily.

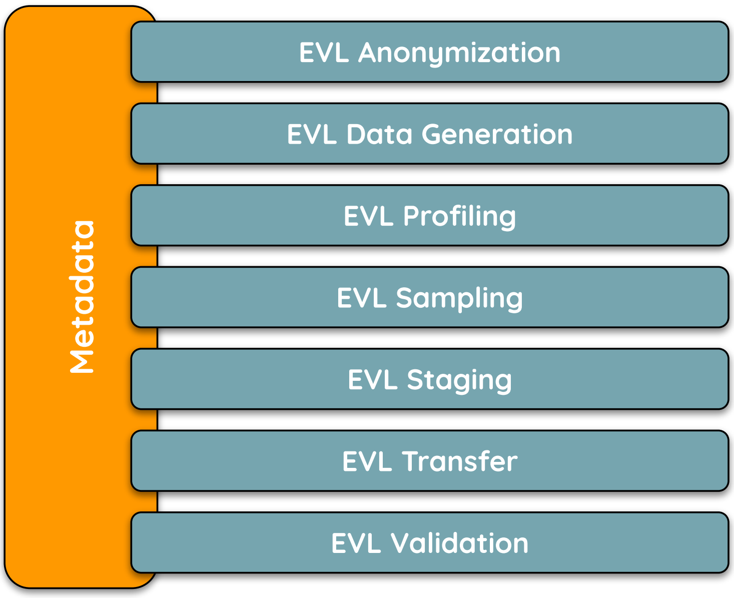

Figure 1.1: EVL Metadata-driven Microservices

All EVL Microservices are built on top of the core **EVL** software and retain its flexibility, robustness, high productivity, and ability to read/write various file formats and databases.

EVL

EVL originally stood for Extract–Validate–Load, but it became a fully featured ETL (Extract–Transform–Load) tool.

EVL is designed with the Unix philosophies of interoperability and "*do one thing, and do it well*" in mind.

Templates, a high level of abstraction, and the ability to dynamically create jobs, make for a powerful ETL tool.

Characteristics

- *Versatile*, i.e. cooperate with other components of the customer's solution, solving only particular problem.
- *High performance*, written in C++.
- *Lightweight*, just install a `rpm`/`deb` package or unzip `tgz`.
- *Highly efficient development* due to strict command-line approach.
- Managed access to the source code (e.g. `git`).
- *Linux only*, using the best of the system.

- Graphical User Interface – EVL Manager.

Features

- Natively read/write[1]:
  - File formats: CSV, JSON, XML, XLS, XLSX, Parquet, Avro, QVD/QVX, ASN.1
  - DBMS: MariaDB/MySQL, Oracle, PostgreSQL, SQLite, ODBC, (near future: Snowflake, Redshift)
  - Cloud storages: Amazon S3 and Google Storage
- Hadoop: read/write HDFS, resolve, build and run Spark jobs, Impala/Hive queries
- Partitioning, to partition data and/or parallelize processing
- Productivity boosters, to generate jobs/workflows from metadata

For the most recent information about EVL and supported formats and DBs please check `https://www.evltool.com`.

EVL Microservices

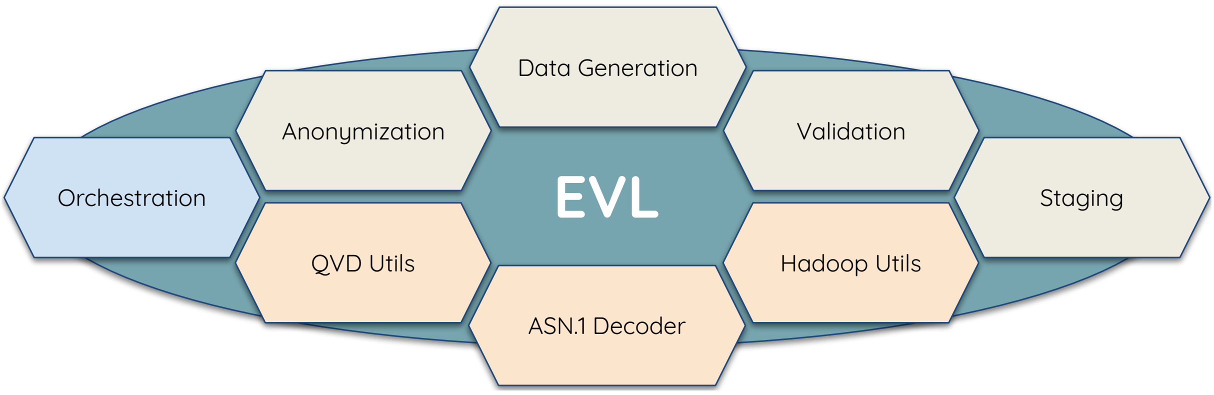

Figure 1.2: EVL Microservices Overview

Each of the Microservice solves particular problem:

Anonymization
: Anonymizing production data according to GDPR requirements and other regulations for developers, testers and outside contractors

Data Generation
: Simulating data complying with the real-life data patterns for proper testing environments, application development and implementing ETL processes

Validation
: Replacing heavy and complex testing tools in migration projects or quick quality checks of production data

Staging
: Getting data from various sources, like Oracle, Teradata, Kafka, CSV or JSON files, and providing a historized base stage

QVD Utils
: Enables reading/writing QVD files without using Qlik Sense or QlikView and also provides metadata from QVD/QVX header.

[1] Actually any source/target can be used, once available from Linux.

Hadoop Utils
: Reading/writing Parquet and Avro file formats, query Impala, Hive, etc.

ASN.1 Decoder
: Decoding files from ASN.1 format into JSON with the highest performance

Orchestration
: Scheduling and monitoring sequences of jobs and workflows, awaiting file delivery, etc. Viewing the workflows in a graphical user interface and starting, restarting, canceling jobs and workflows and checking their statistics and logs.

But they could supply each other, so combining them one get the whole solution.

For the most recent list of EVL Microservices and additional information please visit `https://www.evltool.com`.

2 Release Notes

Versions numbering: EVL Data Generation *x.y.z*

x – major release, i.e. big changes must happen to advance this number

y – minor releases, i.e. introduce new features

z – bugfixes

Overview

[Version 0.1], page 4, (2019/03)
: Initial version of Anonymization Microservice.
EVL version needed: 2.0
New features: Anonymization, Unique Anonymization, Checksum, Salted checksum, Tokenization.

[Version 0.2], page 5, (2019/07)
: Randomization added, dates and timestamps handling improved.
EVL version needed: 2.1
Changes: Function names refined, dates/timestamps '`9999-12-31`' and '`4712-12-31`' are not anonymized by default.
New features: Masking, Randomization, Min and max can be specified also for dates and timestamps.

[Version 1.0], page 5, (2020/01)
: Excel sheet can be used as config, workflow generator introduced.
EVL version needed: 2.2
Changes: project folder structure reorganized, Anon type names get shorter.
New features: EVL workflow generator, project generator.

[Version 1.1], page 6, (2020/05)
: EVL version needed: 2.3
New features: predefined business anonymization types, config file checker

[Version 2.4], page 6, (2020/10)
: Version numbers synchronized with EVL, EVL Manager added – a graphical interface, IBAN anonymization, anonymization by lookup.
New features: business anon type '`ANON_IBAN`', anon function '`ANON_LOOKUP`'.

Version 0.1

Released 2019/03

EVL Version needed
2.0

Description
Initial version

New features
- anonymization,
- unique anonymization,
- checksums,
- salt,
- tokenization

Version 0.2

Released 2019/07

EVL Version needed
2.1

Changes

- Function ‘`TOKENIZE`’ renamed to ‘`ANONYMIZE_UNIQ`’ as it better describes what it does. To migrate, it is enough to replace all in anon-config file(s) and regenerate jobs.
- Function ‘`TOKENIZE_LKP`’ renamed to ‘`TOKENIZE`’. To migrate, it is enough to replace all in anon-config file(s) and regenerate jobs.
- Default Anonymization options can be added into project.sh:

  ```
  export EVL_ANON_DEFAULT_MIN_STRING_LENGTH=0
  export EVL_ANON_DEFAULT_MAX_STRING_LENGTH=10
  ```

  Then for string fields these values are used when ‘`min_length`’ and/or ‘`max_length`’ are empty in anon-config file.
- When running generated anonymization jobs, **data are appended** to the target file/table, not overwritten.
- Dates/timestamps 9999-12-31 and 4712-12-31 are not anonymized by default.

New features

- randomization,
- Min and Max can be specified also for dates and timestamps,
- masking.

Migration script

To migrate your config CSV file from version 0.1, just run:

```
# cd to your project directory
for i in anon-config.*.csv
do
  mv $i $i.bckp
  sed 's/TOKENIZE\([^_]\)/ANONYMIZE_UNIQ\1/;
       s/TOKENIZE_LKP/TOKENIZE/' $i.bckp >$i
done
```

Version 1.0

Released 2020/01

EVL Version needed
2.2

Description

Folder structure reorganized. Next to CSV, also an intelligent Excel config file can be used (list values, syntax checking). Job generators enhanced and generation of EVL workflows added.

Changes

- project folder structure reorganized,
- Anon type names get shorter: ‘`ANONYMIZE`’ changed to ‘`ANON`’.

New features

- an intelligent Excel config file can be used (list values, syntax checking),

- project generator,
- EVL Workflow generator,

Migration script

Create new project in current directory:

```
evl anon new project your_project_name
```

Then modify your configs in old project:

```
# cd to your project directory
for i in anon-config.*.csv
do
  mv $i $i.bckp
  sed 's/;ANONYMIZE/;ANON/' $i.bckp >$i
done
```

and copy them into your new project directory:

```
cp anon.*.csv your_project_name/
```

And copy all your custom settings from ‘`project.sh`’ and ‘`run/generate_jobs.*.evl`’ files.

Version 1.1

Released 2020/05

EVL Version needed
2.3

Description

Basic syntax validation of config file. Configurable anonymization types with the set of predefined ones.

New features

- check config file syntax,
- predefined business anonymization types:
  ‘`ANON_AMOUNT`’, ‘`ANON_EMAIL`’ and ‘`ANON_NAME`’.

Version 2.4

Released 2020/10

Description

Version numbers synchronized with EVL. EVL Manager added – a graphical interface. Anonymize by lookup.

New features

- predefined anonymization functions for using lookups: ‘`ANON_LOOKUP`’,
- IBAN anonymization predefined types:
  ‘`ANON_IBAN`’, ‘`ANON_IBAN_KEEP_COUNTRY`’, ‘`ANON_IBAN_KEEP_BANK`’,
- GUI: integration with EVL Manager,
- Next to Oracle, PostgreSQL, Teradata, also MySQL/MariaDB can be connected.

Version 2.5

Plan to release
2021/04

New features

- Files can be read/written from/to Samba, HDFS, SFTP, AWS S3 and Google Storage.
- Config file generator also for JSON files and for Oracle and PostgerSQL tables.

3 Installation and Settings

Trial Installation

- Section 3.1 [Linux RPM], page 8, – installation on RedHat-like systems,
- Section 3.2 [Linux DEB], page 8, – installation on Debian-like systems,
- Section 3.3 [Windows], page 8, – installation on Windows 10.

Enterprise Installation

- Section 3.4 [Linux RPM], page 9, – installation on RedHat-like systems,
- Section 3.5 [Other Unix Systems], page 9, – installation on other Linuxes, MacOS, etc.

Settings

- Section 3.6 [Settings], page 9, – initial set up.

3.1 Trial – Linux RPM

I.e. RedHat, CentOS, Fedora, Oracle Linux.

Get the package for your OS from https://www.evltool.com/downloads and

```
tar xvf evl-data-generation-trial.*.rpm.tar
sudo dnf install ./evl-common-2.8*.x86_64.rpm
sudo dnf install ./evl-core-2.8*.x86_64.rpm
sudo dnf install ./evl-data-generation-2.8*.x86_64.rpm
```

Then initiate the installation for current user by

```
/opt/evl/bin/evl --init
```

3.2 Trial – Linux DEB

I.e. Ubuntu, Debian, etc.

Get the package from https://www.evltool.com/downloads and

```
tar xvf evl-data-generation-trial.ubuntu.deb.tar
sudo apt install ./evl-common_2.8*_amd64.deb
sudo apt install ./evl-core_2.8*_amd64.deb
sudo apt install ./evl-data-generation_2.8*_amd64.deb
```

Then initiate the installation for current user by

```
/opt/evl/bin/evl --init
```

3.3 Trial – Windows

There is no native package of EVL Data Generation for Microsoft Windows, however Windows Subsystem for Linux (WSL) might be used on Windows 10. Data Generation can run on Ubuntu under Windows, while all files (either data or configurations) can be edited from Windows.

Ubuntu 18.04 can be installed from Microsoft Store.

Windows Terminal Preview is not necessary, but make the work with Ubuntu easier. Again can be installed from Microsoft Store

Then continue like installing DEB package on Ubuntu.

3.4 Enterprise – Linux RPM

I.e. RedHat, CentOS, Fedora, Oracle Linux.

Having enterprise version you obtain the credentials to RPM repository, so then for example for CentOS or Oracle Linux of version 8 it would be standard way to install a package:

```
sudo dnf install evl-data-generation
```

Then initiate the installation for current user by

```
/opt/evl/bin/evl --init
```

3.5 Enterprise – Other Unix Systems

Basically any standard Unix system with Bash and couple of standard utilities (gettext, binutils, coreutils) and libraries (zlib, ICU, xml2) is possible. Ask `support@evltool.com` for help.

3.6 Settings

Enterprise version of EVL Data Generation (and also possibly other EVL Microservices) resides in `/opt/evl`. To initiate EVL for current user, run

```
/opt/evl/bin/evl --init
source $HOME/.evlrc
```

which adds an `.evlrc` file into your `$HOME` folder and adds sourcing it into `$HOME/.bashrc`.

Then one can check, add or modify several settings in `.evlrc`, for example variable `EVL_ENV`. These settings are top level settings for given user. (Later there are `project.sh` files in each project, to set project-wide variables.)

After that EVL Data Generation is ready to use. Good to start is to create a new project with some sample data, jobs and workflows:

```
evl anon project sample my_anon_sample
```

Compiler

Data Generation mappings are compiled either by GCC or Clang. Which one is used depends on environment variable `EVL_COMPILER`, these two values are possible:

```
EVL_COMPILER=gcc
EVL_COMPILER=clang
```

If this variable is not set, then on Linux systems is GCC by default, and on Windows and Mac it is Clang.

GCC must be at least in the version 7.4 and Clang at least 6.0.

4 Data Generation

Before we will go into detail, let's provide an overview of data generation process.

To initiate, setup and build a project (i.e. group of data you would like to anonymize) follow these steps. See Section 4.1 [evl datagen command], page 10, for details about '`evl datagen`' commands.

1. Create new project

   ```
   evl datagen project new <project_dir>
   ```

   See Section 4.2 [Project], page 12, for details about projects.
2. Add a source, i.e. folder with files to be anonymized or database with tables to be anonymized:

   ```
   evl datagen source new <source_name> \
     --guess-from-csv <path_to_folder_with_such_CSVs>
   ```

   See Section 4.3 [Source Settings], page 13, for details about settings for a source.
3. Edit such a config (`csv`) file according to your preferences. (Excel file checks the validity immediately and provides drop down options.)
4. Check the config file for mistakes

   ```
   evl datagen check <config_file>
   ```
5. Generate anonymization jobs and workflow

   ```
   evl datagen build <config_file>
   ```

   See Section 4.4 [Build and Run], page 13, for details about jobs and workflow generation and see Chapter 5 [Config File], page 15, for details about a config file.

Then to anonymize (regularly), run anonymization jobs:

```
evl run/datagen/<table_1>.evl
evl run/datagen/<file_1>.evl
...
```

Each job represents one file or table to be anonymized. See Section 4.4 [Build and Run], page 13, for details.

> **Note:** Be careful running anonymization jobs several times, as data are by default **overwritten** in the target, unless `export EVL_DATAGEN_APPEND=1` is specified in settings `configs/datagen/*.sh` file or `project.sh`.

See [Environment variables], page 12, for details about all possible configuration `EVL_DATAGEN_*` variables.

Having many files or tables to anonymize in one batch, you don't need to run anonymization jobs one after another, but you can run all jobs by running generated workflow:

```
evl run workflow/datagen/<source_name>.ewf
```

4.1 evl datagen command *(since EVL 1.0)*

To help to generate, check and build all the configuration files, there is '`evl datagen`' command line utility.

`evl datagen project new <project_dir>`
: creates new project folder `<project_dir>` with default folder structure and files inside.

`evl datagen project sample <project_dir>`
: creates new project folder `<project_dir>` with sample data and configs.

`evl datagen source new <source_name>`
: creates new source `<source_name>` in current project directory (or in `<project_dir>`). With '`--guess-from-csv`' option, it guess data types based on source csv files.

`evl datagen check <config_file>`
: check if `<config_file>` contains valid combination of metadata.

`evl datagen build <config_file>`
: generates data-generation jobs based on `<config_file>` and also a Workflow with all these jobs.

Synopsis

```
evl datagen project
  ( new | sample ) <project_dir>
  [-v|--verbose]

evl datagen source new
  <source_name>
  [-p|--project <project_dir>]
  [-g|--guess-from-csv <source_dir>]
  [-v|--verbose]

evl datagen check
  <config_file>
  [-p|--project <project_dir>]
  [-v|--verbose]

evl datagen build
  <config_file>
  [-p|--project <project_dir>]
  [--parallel [<parallel_threads>]]
  [-v|--verbose]

evl datagen
  ( --help | --usage | --version )
```

Options

`-p, --project=<project_dir>`
: if the current directory is not a project's one, full or relative path can be specified by `<project_dir>`

`--parallel[=<parallel_threads>]`
: generate workflow with jobs parallelized as much as possible. To limit this parallelization to, `<parallel_threads>` can be specified, which is the value how many jobs can run in parallel.

`-g, --guess-from-csv=<source_dir>`
: preserve mode (i.e. permission), timestamps and ownership

Standard options:

`--help`
: print this help and exit

`--usage`
: print short usage information and exit

`-v, --verbose`
: print to stderr info/debug messages of the component

`--version`
: print version and exit

Environment Variables

The list of all EVL Data Generation variables with their default values. One can change these values in his '`~/.evlrc`' file or in the project in '`project.sh`'.

`EVL_DATAGEN_APPEND=0`
: whether append or overwrite target files/tables. Possible values are '`0`' or '`1`'.

`EVL_DATAGEN_EOL=""`
: whether Linux ('`\n`'), Windows ('`\r\n`') or old Mac ('`\r`') end-of-lines. Possible values are "dos", "mac", or leave empty for Linux EOL.

`EVL_DATAGEN_HEADER=1`
: whether or how many lines has file header. Zero means no header.

`EVL_CONFIG_EOL=""`
: whether Linux ('`\n`'), Windows ('`\r\n`') or old Mac ('`\r`') end-of-lines are used for main config CSV file. Possible values are "dos", "mac", or leave empty for Linux EOL.

`EVL_CONFIG_FIELD_SEPARATOR=";"`
: the default field separator used in config files

`EVL_DEFAULT_FIELD_SEPARATOR=";"`
: the default field separator for CSV files. This character might be any one of the first 128 ascii ones.

`EVL_DEFAULT_RECORD_SEPARATOR='\n'`
: the default record separator for CSV files. This character might be any one of the first 128 ascii ones. By default a Linux newline is used. To use Windows end of line (i.e. '`\r\n`'), use '`EVL_DATAGEN_EOL`' variable

4.2 Project

Consider an anonymization project to be a folder, where we work on anonymization of some group of data. For example a group of data from business point of view. In most cases there would be only one or a couple of projects.

You can create a new project by hand or by a command:

```
evl datagen project new my_project
```

It will create new directory `my_project` in current folder with default settings and subfolder structure.

Or you can a new project with sample data and configuration:

```
evl datagen project sample $HOME/my_sample_project
```

It will create new directory `my_sample_project` in your home folder with a sample project.

The anonymization project directory structure is:

`build/`	files generated by '`evl datagen build`' command
`configs/`	configuration `csv` files and settings `sh` files
`lib/`	folder for custom anonymization functions
`run/`	anonymization jobs generated by '`evl datagen build`' command
`worflow/`	workflows generated by '`evl datagen build`' command

All files in `build`, `run` and `workflow` directories are completely generated based on configuration file(s) `configs/<source_name>.csv`.

4.3 Source Settings

Once we have a project directory, we would like to add a source, which could be a folder with files or a database.

What and how should be anonymized is specified in a **config** and **setting** files. Config file could be a `csv` file and setting file is a shell script with variables definitions.

Each source would have one config and one setting file.

To create a **new empty config** and setting files, run:

```
evl datagen source new my_source
```

which creates two files in current project folder

```
configs/my_source.csv
configs/datagen/my_source.sh
```

To create a **pre-generated config** and setting files, based on a folder with source `csv` files:

```
evl datagen source new my_source --guess-from-csv=data/source
```

which goes through all `csv` files in `data/source` folder and fill in config file entity names (i.e. file names), field names based on headers, data types and null flag of a field.

If the current directory is not the project's one, specify the path to the project by option '`--project=<project_path>`'.

See Chapter 5 [Config File], page 15, for detailed information about config files.

4.4 Build and Run

For each Entity from config file, i.e. table or file, anonymization job with mapping and other metadata need to be build. It is enough to run the command line utility

```
evl datagen build <config_file>
                  [-p|--project <project_dir>]
                  [--parallel [<parallel_threads>]]
                  [-v|--verbose]
```

That build all the files in `build/` project subdirectory. There you can find `evd` and `evm` files in appropriate folders. EVD means *EVL Data definition* file and it defines the structure of the source/target; field names, data types and other attributes. EVM means *EVL Mapping* file and it defines how each field is mapped. Although both these files are generated, it is sometimes good to check how they are look like for debug purpose.

It generates also a file in `run/datagen/` subdirectory, where you can find one `evl` file per each Entity. These files can be then run to anonymize the data. For example for three tables, `party_addr`, `party_cont` and `party_rel` it would be fired by these commands:

```
evl run/datagen/party_addr.evl
evl run/datagen/party_cont.evl
evl run/datagen/party_rel.evl
```

Once such `evl` file exists for an Entity, there no need to build jobs again. It check each run if the config file has changed or not for given Entity and run '`evl datagen build`' command automatically.

> **Note:** There is no need to run '`evl datagen build`' every time the config file is updated. It is done automatically once the job is fired.

The build command also generates a workflow file for given source in `workflow/datagen/` subdirectory. You can run the anonymization for all the Entities from that source. For example having source defined by `configs/some_source.csv`, you can run

```
evl run workflow/datagen/some_source.ewf
```

and it will run all anonymization jobs in one or several parallel threads. It depends on the value defined by `--parallel` option.

If one or more anonymization jobs in a workflow fail, then you can the restart the whole workflow by:

```
evl restart workflow/datagen/some_source.ewf
```

or continue from those last failures:

```
evl continue workflow/datagen/some_source.ewf
```

5 Config File

Main configuration file defines information about each field to be processed. It is shared to all metadata-driven EVL Microservices, so it is situated in `configs` project directory. It is a `csv` file named by a source, so you can have several config files in a project, for example:

```
configs/mssql_exports.csv
configs/oracle_db1_tables.csv
configs/oracle_db2_tables.csv
configs/some_json_source.csv
```

Here is an example of the most of the columns of such configuration `csv` file (semicolon delimited by default):

src_type	**entity_name**	**order**	**field_name**	**data_type**	**null**	**anon_type**	**evl_value**
ORA	accounts		id	int	No	ANON_UNIQ	
ORA	accounts		cust_id	int	No	ANON_LOOKUP	
ORA	accounts		iban	string		ANON_IBAN	
ORA	accounts		currency	string			
ORA	accounts		score	decimal(8,2)		ANON_AMOUNT(0.1)	
ORA	accounts		when	date		ANON_VAR	
FILE	cust.csv	1	id	int	No	ANON_UNIQ	
FILE	cust.csv	2	email	string		ANON_EMAIL	
FILE	cust.csv	3	pers_id	string	No		anon_ rc(IN)

Description of each column of such configuration file follows.

5.1 Source Type

is used to distinguish different inputs. Possible values are:

`FILE, file`
: for any supported file format, e.g. `avro`, `csv`, `json`, `parquet`, `qvd`, `xls`, `xlsx`, `xml`. To distinguish a file format, file suffix is used from the Entity name.

`avro, csv, json, parquet, qvd, xls, xlsx, xml`
: to be used only in the case the file name or mask (i.e. an Entity name) has no proper file suffix.

`MySQL` for MySQL/MariaDB tables

`ORA, Oracle`
: for Oracle tables

`PG, PostgreSQL`
: for PostgreSQL tables

`TD, Teradata`
: for Teradata tables

When the field '`source_type`' is empty, then option '`FILE`' is supposed.

5.2 Entity Name

is mandatory and it is either *file name/mask* or *table name.*

In case of file sources the file name or mask should contain proper file extension to be distinguished. It supports also compression suffixes `.gz`, `.tar.gz`, and `.zip`, and process them properly.

5.3 Field Order

is mandatory only for `csv` files. It specifies the order of given field in a file, counted from 1.

Specifying the order for `json`, `qvd`, `xls`, `xlsx`, or `xml` files will omit using option `--match-fields` by EVL Read component which increase speed of the anonymization.

5.4 Field Name

is simply a field name. Case sensitive!

Field name is also used for mapping generation, so it appears in EVM mapping file and can be used in '`evl_value`' configuration field.

5.5 EVL Datatype

Possible EVL data types are:

`string`
: to be used for all texts, varchars, etc.

`char, uchar, short, ushort, int, uint, long, ulong, int128, uint128`
: to be used for all corresponding integral data types

`decimal`
: to be used for `decimal(m,n)` and `number(m,n)` data types

`float, double`
: to be used for `float`, `double`, `real` and similar source data types

`date, datetime, timestamp`
: to be used for corresponding date and time data types, where `date` keeps only date, `datetime` keeps date and time (in seconds), and `timestamp` can hold also nanoseconds and time zone.

5.6 Format

To specify a format for date and time data types and decimal places for decimals.

For `decimal` data type you must specify '`m,n`', where '`m`' is number of all digits and '`n`' is the number of decimal places.

For date and time data types, when no format is specified, defaults are used:

`EVL_DEFAULT_DATE_PATTERN="%Y-%m-%d"`
: to specify default formatting string for '`date`' data type

`EVL_DEFAULT_TIME_PATTERN="%H:%M:%S"`
: to specify default formatting string for '`time`' data type

`EVL_DEFAULT_DATETIME_PATTERN="%Y-%m-%d %H:%M:%S"`
: to specify default formatting string for '`datetime`' data type

`EVL_DEFAULT_TIMESTAMP_PATTERN="%Y-%m-%d %H:%M:%E*S"`
: to specify default formatting string for '`timestamp`' data type

To specify a formatting pattern, standard C notation can be used. Examples of such formatting strings:

`%d.%m.%Y`
: to have a date like '`21.01.2026`'

`%d-%b-%Y`
: to have a date like '`21-Jan-2026`'

`%Y%m%d%H%M%S`
: to have a datetime like '`20260121093000`'

`%Y-%m-%dT%H:%M:%S.%E3f`
: to have a timestamp like '`2026-01-21T09:30:59.545`'

5.7 Nullable

says if the field is nullable or not. When empty, it suppose the field is nullable. Possible values are:

`1, Y, y, yes, Yes, True, true`
: for NULLABLE

`0, N, n, no, No, False, false`
: for NOT NULL

For file sources, when the field is NULLABLE, then an empty string is interpreted as NULL and anonymization functions keep such field again NULL.

> **Important:** When a field is flagged as NOT NULL, then an empty string is manipulated as proper string, so anonymization functions anonymize them. In such case from an empty string one can get non-empty anonymized value.

5.8 Min and Max

Strings

For strings it specifies, how long strings it should create on the output when anonymize.

By default it supposes '`min`' = 0 and '`max`' = 10, so for string fields of the length less than 10, it is actually mandatory to set '`max`' length. To change these defaults, one can specify other values in `project.sh` configuration file.

```
export EVL_DATAGEN_DEFAULT_MIN_STRING_LENGTH=0
export EVL_DATAGEN_DEFAULT_MAX_STRING_LENGTH=10
```

Numbers

For numbers it contains the range of values it should produce when doing anonymization. Here are environment variables with their default values (min/max values from config file take precedence):

```
export EVL_DATAGEN_DEFAULT_MIN_CHAR=0
export EVL_DATAGEN_DEFAULT_MIN_SHORT=0
export EVL_DATAGEN_DEFAULT_MIN_INT=0
export EVL_DATAGEN_DEFAULT_MIN_LONG=0
export EVL_DATAGEN_DEFAULT_MIN_DECIMAL=0
export EVL_DATAGEN_DEFAULT_MIN_FLOAT=0
export EVL_DATAGEN_DEFAULT_MIN_DOUBLE=0
export EVL_DATAGEN_DEFAULT_MAX_CHAR=99
export EVL_DATAGEN_DEFAULT_MAX_SHORT=9999
export EVL_DATAGEN_DEFAULT_MAX_INT=99999999
export EVL_DATAGEN_DEFAULT_MAX_LONG=99999999
export EVL_DATAGEN_DEFAULT_MAX_DECIMAL=99999999
export EVL_DATAGEN_DEFAULT_MAX_FLOAT=1000000
export EVL_DATAGEN_DEFAULT_MAX_DOUBLE=1000000
```

Dates and times

For dates and times it contains the range it should produce when calling `anonymize(value,min,max)` function (e.g. by `ANON` anon type). Default values for date, datetime and timestamp are set by environment variables to '`1970-01-01`' and '`2099-12-31`' this way:

```
export EVL_DATAGEN_DEFAULT_MIN_DATE="1970-01-01"
export EVL_DATAGEN_DEFAULT_MIN_DATETIME="1970-01-01 00:00:00"
export EVL_DATAGEN_DEFAULT_MIN_TIMESTAMP="1970-01-01 00:00:00.000000000"
export EVL_DATAGEN_DEFAULT_MAX_DATE="2099-12-31"
export EVL_DATAGEN_DEFAULT_MAX_DATETIME="2099-12-31 00:00:00"
export EVL_DATAGEN_DEFAULT_MAX_TIMESTAMP="2099-12-31 00:00:00.000000000"
```

Note: All the environment variables can be defined either in `project.sh` (for the whole project), or in `configs/datagen/<source_name>.sh` to have it only for sources given by `<source_name>.csv`.

5.9 Anon Type

Following predefined types of anonymization can be specified in the '`anon_type`' configuration column.

`ANON` this type is applicable for all EVL data types. It uses '`anonymize(value, max, min)`' EVL function for given data type. It takes min/max value from columns '`min`' and '`max`' if such exist, otherwise it uses default values from variables starts with `EVL_DATAGEN_DEFAULT_MAX_`.

In general it may happen that for two different inputs it produces the same anonymized value. So it may happen that the mapping is not bijection, which is mostly the good way how to anonymize.

Important: For given value and given salt it produces always the same anonymized value.

`ANON_VAR` can be used only for date and time data types. Internally it uses '`anonymize(value, days, month, years)`' for date, '`anonymize(value, seconds, minutes, hours, days, month, years)`' for datetime, and '`anonymize(value, nanoseconds, seconds, minutes, hours, days, month, years)`' for timestamp. Variance arguments are taken from following environment variables, which are set by default:

```
export EVL_DATAGEN_DEFAULT_NANOSECONDS=500000000
export EVL_DATAGEN_DEFAULT_SECONDS=30
export EVL_DATAGEN_DEFAULT_MINUTES=30
export EVL_DATAGEN_DEFAULT_HOURS=12
export EVL_DATAGEN_DEFAULT_DAYS=15
export EVL_DATAGEN_DEFAULT_MONTHS=6
export EVL_DATAGEN_DEFAULT_YEARS=5
```

`ANON_UNIQ`

can be used only for integral data types. Internally it uses '`anonymize_uniq()`' EVL function. It produces the output in a unique way, so bijection is guaranteed. Particularly useful for IDs.

Important: It keeps the mapping to be a bijection.

`ANON_NAME`

can be used only for string data type and internally maps to '`anonymize(<IN>," ", true)`', i.e. keep the length, spaces, capital letters will be again capitals, lowercase letters stay lowercased and numbers will be numbers again.

`ANON_EMAIL`

can be used only for string data type and internally maps to '`anonymize(<IN>,"@.")`', i.e. keep '`@`' and dots unchanged, other parts are anonymized as usual. Length is not kept.

`ANON_IBAN`
`ANON_IBAN_KEEP_COUNTRY`
`ANON_IBAN_KEEP_BANK`

applicable only for strings and maps into

```
anonymize_iban(<IN>)
anonymize_iban(<IN>,iban_anon::keep_country)
anonymize_iban(<IN>,iban_anon::keep_country_and_bank)
```

So in the first case it anonymize IBAN into arbitrary country IBAN, but keep the validity. In second case it keeps the country, and in third one also keeps the bank.

`ANON_AMOUNT()`

is applicable for `int`, `uint`, `long`, `ulong` and `decimal`. It is defined as

```
anonymize(<*IN>, <*IN> - <ARG> * <*IN>, <*IN> + <ARG> * <*IN>)
```

where '`<*IN>`' is placeholder for the input value and '`<ARG>`' is an argument specified in the parentheses. So for example for '`ANON_AMOUNT(0.1)`' returns anonymized value within plus/minus 10% interval.

`MASK_LEFT()`
`MASK_RIGHT()`

can be used only for strings and it calls '`str_mask_left(<IN>,<ARG>)`' or '`str_mask_right(<IN>,<ARG>)`', where '`<IN>`' is placeholder for the pointer to the input value and '`<ARG>`' is an argument specified in the parentheses. So for example '`MASK_LEFT(4)`' masks the string by '`*`' from left, but keep 4 characters unchanged.

`RANDOM` behaves very similar way as '`ANON`' type, but each run may return different value. Applicable for any data type; internally calls '`random_<data_type>(min,max)`', where min/max is taken from config file (see [Min and Max], page 17) or is defined by environment variables.

`RANDOM_VAR`

behaves very similar way as '`ANON_VAR`' type, but each run may return different value. Applicable only for date and time data types. It calls internally '`randomize()`' function.

`ANON_LOOKUP`

firstly it creates a lookup from the column and then use it as possible values. This way you get real values, but shuffled.

> **Important:** For given value and given salt it produces always the same anonymized value.

`ANON_LOOKUP()`

it uses some already prepared lookup specified as an argument in parentheses. So anonymized values are always taken from this lookup. Such a lookup must be prepared by this EVD

```
/opt/EVL-2.8/share/templates/anonymization/datagen/evd/lookup.string.evd
```

which defines data structure:

```
key int
value string null=""
```

and must be sorted by '`key`' field and starts with zero and incremented by 1.

> **Important:** For given value and given salt it produces always the same anonymized value.

`RANDOM_LOOKUP`

behaves very similar way as '`ANON_LOOKUP`' type, but each run may return different value.

`RANDOM_LOOKUP()`
: behaves very similar way as '`ANON_LOOKUP()`' type, but each run may return different value.

`TOKEN*`
: group of anonymization types starts with '`TOKEN`' are not really defined, but act exactly like all above types starts with '`ANON`'. The only difference is that it keeps the conversion table in the `$EVL_DATAGEN_TOKEN_DIR` directory, so you can revert the anonymized values if needed. Once some value is anonymized and so presented in the conversion table, the same value will be again anonymized the same way, no matter if salt has changed at meantime.

  > **Important:** This anonymization type is revertible, but that is the purpose.

All of these anonymization types are predefined in

`/opt/EVL-2.8/share/templates/anonymization/configs/datagen/anon-functions-config.csv`

It is actually a mapping of anonymization type and data type to exact internal EVL (or custom C/C++) function. For example '`MASK_LEFT()`' is defined by

```
anon_type;evl_datatype;evl_value;description
MASK_LEFT();string;str_mask_left(<IN>,<ARG>);
```

Custom Anon Types

You can also define your own custom anonymization types within your Anonymization project by specifying in your project folder a file `configs/datagen/anon-functions-config.csv` of the same structure. You can define for example anonymization type '`MASK_LEFT_X()`', which mask input from left by '`X`', but keep `<ARG>` characters unchanged, this way:

```
anon_type;evl_datatype;evl_value;description
MASK_LEFT_X();string;str_mask_left(<IN>,<ARG>,'X');
```

Placeholders

Following possible placeholders can be used in Anonymization Type definition file:

`<IN>`
`<*IN>`
: to be replaced by the value of the input field ('`<*IN>`') or by the pointer to that value ('`<IN>`')

`<ARG>`
: for Anon Types ended with '`()`' it will resolve to the content specified in the parentheses

`<MIN>`
`<MAX>`
: to be replaced by values from min/max fields from config file or by default values specified by `EVL_DATAGEN_DEFAULT_MIN_*` and `EVL_DATAGEN_DEFAULT_MAN_*` environment variables

`<FIELD_NAME>`
: to be replaced by values a field name

`<NANOSECONDS>`
`<SECONDS>`
`<MINUTES>`
`<HOURS>`
`<DAYS>`
`<MONTHS>`
`<YEARS>`
: to be replaced by values specified by environment variables: `EVL_DATAGEN_DEFAULT_NANOSECONDS`, `EVL_DATAGEN_DEFAULT_SECONDS`, `EVL_DATAGEN_DEFAULT_MINUTES`,

EVL_DATAGEN_DEFAULT_HOURS, EVL_DATAGEN_DEFAULT_DAYS, EVL_DATAGEN_DEFAULT_MONTHS, and EVL_DATAGEN_DEFAULT_YEARS

5.10 EVL Value

> **Important:** When also 'anon_type' column has a value in the config file, then this 'evl_value' is applied first and then also an anonymization type

This field can contain an EVL mapping function(s) to be applied on the given fields. This can be used for special cases, when you don't want to create new custom anonymization type or when you need to somehow prepare a field first.

For example when you want to achieve some business logic to be kept. Like some date must be greater than some other one:

field_name	**evl_datatype**	**anon_type**	**evl_value**
birth_date	date	ANON	
death_date	date		anonymize(IN, *out->birth_date+1, *out->birth_date+36500)

In this example the anonymized date of death will be always in between the anonymized birth date plus one day and anonymized birth date plus 100 years (36500 days).

5.11 Ignored Values

Next to NULLs, there are also usually several other values which you'd like to keep unchanged by anonymization. For example dates like '1970-01-01', '4712-12-31', or '9999-12-31'.

For such purpose, these environment variables can be defined either in `project.sh` (for the whole project), or in `configs/datagen/<source_name>.sh` to have it only for sources given by `<source_name>.csv`:

`EVL_DATAGEN_IGNORE_DATE`
`EVL_DATAGEN_IGNORE_DATETIME`
`EVL_DATAGEN_IGNORE_TIMESTAMP`
: can be empty or contains a comma separated list of values to be ignored when given data type is used,

`EVL_DATAGEN_IGNORE_STRING`
: can be empty or contains a comma separated list of values to be ignored when string is used,

`EVL_DATAGEN_IGNORE_CHAR`
`EVL_DATAGEN_IGNORE_SHORT`
`EVL_DATAGEN_IGNORE_INT`
`EVL_DATAGEN_IGNORE_LONG`
`EVL_DATAGEN_IGNORE_DECIMAL`
`EVL_DATAGEN_IGNORE_FLOAT`
`EVL_DATAGEN_IGNORE_DOUBLE`
: can be empty or contains a comma separated list of numbers to be ignored when given data type is used. Unsigned variants for integral data types use their signed ones.

Here is an usual example for keeping dates unchanged when anonymize database tables.

```
export EVL_DATAGEN_IGNORE_DATE="1970-01-01,4712-12-31,9999-12-31"
export EVL_DATAGEN_IGNORE_DATETIME="1970-01-01 00:00:00,4712-12-31 00:00:00"
```

6 EVL Functions

As an 'EVL value' in anon-config file, arbitrary EVL functions and expressions can be used. An input field is represented as 'IN'.

All functions can be used in two ways:

- with pointers (preferred)
- without pointers (i.e. as referenced values, "with star")

Option with pointers is preferred as it can handle NULL values ('nullptr' in fact). So these two examples:

```
str_function(IN)
str_function(*IN)
```

are basically the same, but the first one might fail in case of using some standard C++ function and NULL value arrive. All EVL functions handle NULLs and (mostly) returns also NULL, so use 'IN' for them. In all other cases use better '*IN'.

There are these two rules in all EVL string manipulation functions described in this section:

- When the first argument is a pointer, the function returns also a pointer.
- When the first argument is 'nullptr', the function returns 'nullptr' as well.

6.1 Randomization Functions

For randomization functions are used same rules regarding 'nullptr' as for string functions.

`randomize()` *(since EVL 2.1)*

Examples:

```
// random int from whole int range
out->random_int       = randomize(in->value);
// random int from interval < value - 1000 , value + 2000 >
out->random_int_range = randomize(in->value,-1000,2000);
```

`random_int()`
`random_long()`
`random_short()`
`random_char()` *(since EVL 2.1)*

Examples:

```
// random value from whole int range
out->random_value = random_int();
// random value from interval <1000,2000>
out->random_range = random_int(1000,2000);
```

`random_float()`
`random_double()` *(since EVL 2.1)*

Examples:

```
// random value from whole float range
out->random_value = random_float();
// random float value from interval <1000,2000>
out->random_range = random_float(1000,2000);
```

`random_decimal()` *(since EVL 2.1)*

Examples:

```
// random value from whole decimal range
out->random_value = random_decimal();
// random float value from interval <1000,2000>
out->random_range = random_decimal(1000,2000);
```

`random_date()`
`random_datetime()`
`random_timestamp()` *(since EVL 2.1)*

Examples:

```
// random date between 1970-01-01 and 2069-12-31
out->random_value = random_date();
// random date from this century
out->random_range = random_date(date("2000-01-01"), date("2099-12-31"));
```

`random_string()` *(since EVL 2.1)*

Examples:

```
// random string of length between 0 and 10
out->random_value = random_string();
// random string of length 5
out->random_range = random_string(5,5);
```

6.2 String Functions

All string manipulation functions can be used in two ways:

- with pointers (preferred)
- without pointers (i.e. as referenced values, "with star")

Option with pointers is preferred as it can handle NULL values (‘`nullptr`’ in fact). So these two examples:

```
out->field  = str_function(in->field);
*out->field = str_function(*in->field);
```

are basically the same, but the second one will fail in case ‘`in->field`’ will be NULL (i.e. ‘`nullptr`’).

There are these two rules in all string manipulation functions described in this section:

- When the first argument is a pointer, the function returns also a pointer.
- When the first argument is ‘`nullptr`’, the function returns ‘`nullptr`’ as well.

6.2.1 length *(since EVL 2.0)*

Returns the length of given string.

For ‘`nullptr`’ it returns again ‘`nullptr`’.

Example:

```
length((string)"Some text")     // return 9
length(nullptr)                 // return nullptr
```

In mapping it might look like this (without pointers):

```
out->str_len = length(in->first_name);
```

6.2.2 split *(since EVL 1.3)*

Example:

```
split("Some text, another text.", ' ')
     // returns vector ["Some", "text,", "another", "text."]
```

When the first argument is ‘`nullptr`’, it returns ‘`nullptr`’.

In mapping it might look like this (without pointers):

```
static std::vector<std::string> name_vec;

name_vec = split(*in->full_name", ' ');
*out->first_name = name_vec[0];
*out->last_name  = name_vec[1];
```

or (preferably) using pointers:

```
static std::vector<std::string*>* name_vec;

name_vec = split(in->full_name", ' ');
out->first_name = name_vec[0];
out->last_name  = name_vec[1];
```

Function headers:

```
std::vector<std::string>   split(const std::string& str, \
                                 const char delimiter);
std::vector<std::string*>* split(const std::string* const str, \
                                 const char delimiter);
```

6.2.3 starts_with, ends_with *(since EVL 2.0)*

True if a string starts or ends with the given substring.

When the first argument is 'nullptr', it returns False.

Example:

```
starts_with("Some text", "Some")   // return True
starts_with("Some text", "x")      // return False
starts_with(nullptr, "x")          // return False
ends_with("Some text", "ext")      // return True
ends_with("Some text", "x")        // return False
```

In mapping it might look like this:

```
*out->test_field = starts_with(in->test_field ? "OK" : "NOK" ;
```

Function headers:

```
bool starts_with(const std::string& str, const char* const prefix);
bool starts_with(const std::string* const str, const char* const prefix);
bool starts_with(const std::string& str, const std::string& prefix);
bool starts_with(const std::string* const str, const std::string& prefix);
bool ends_with(const std::string& str, const char* const suffix);
bool ends_with(const std::string* const str, const char* const suffix);
bool ends_with(const std::string& str, const std::string& suffix);
bool ends_with(const std::string* const str, const std::string& suffix);
```

6.2.4 str_compress, str_uncompress *(since EVL 2.0)*

Compress/uncompress the given string. Examples which return pointers:

```
str_compress(in->string_field_to_compress)       // snappy by default
str_compress(in->string_field_to_compress, compression::gzip)
str_compress(in->snappy_field)                   // snappy by default
str_compress(in->gzipped_field, compression::gzip)
```

Examples which return string values:

```
str_compress(*in->string_field_to_compress)      // snappy by default
str_compress(*in->string_field_to_compress, compression::gzip)
str_compress(*in->snappy_field)                  // snappy by default
str_compress(*in->gzipped_field, compression::gzip)
```

When the first argument is ‘`nullptr`’, it returns ‘`nullptr`’.

In mapping it might look like this:

```
out->gzipped_field = str_compress(in->string_field);
```

Function headers:

```
std::string str_compress(const std::string& str, \
             const compression method = compression::snappy);
std::string* str_compress(const std::string* const str, \
             const compression method = compression::snappy);
```

```
std::string str_uncompress(const std::string& str, \
             const compression method = compression::snappy);
std::string* str_uncompress(const std::string* const str, \
             const compression method = compression::snappy);
```

6.2.5 str_count *(since EVL 1.3)*

It counts the number of occurrences of given string or character. Example:

```
str_count("Some text, another text.", ' ')     // returns 3
str_count("Some text, another text.", "text")  // returns 2
```

When the first argument is ‘`nullptr`’, it returns ‘`nullptr`’.

In mapping it might look like this (using pointers):

```
out->jan_cnt  = str_count(in->first_name", "Jan");
```

or without pointers:

```
*out->jan_cnt = str_count(*in->first_name", "Jan");
```

Function headers:

```
std::size_t  str_count(const std::string& str, const char ch);
std::size_t* str_count(const std::string* const str, const char ch);
std::size_t  str_count(const std::string& str, const char* const substr);
std::size_t* str_count(const std::string* const str, \
                       const char* const substr);
std::size_t  str_count(const std::string& str, const std::string& substr);
std::size_t* str_count(const std::string* const str, \
                       const std::string& substr);
```

6.2.6 str_index, str_rindex *(since EVL 2.0)*

`str_index(str,substr)`
: it returns the index (counted from 0) of the first occurrence of the given substring,

`str_rindex(str,substr)`
: it returns the index (counted from 0) of the last occurrence of the given substring.

When no match, then ‘`-1`’ is returned.

When the string is ‘`nullptr`’, it returns ‘`nullptr`’.

Examples:

```
str_index("Some text text", "text")   // return 5
str_index("Some text text", "xyz")    // return -1
str_index(nullptr, 'x')              // return nullptr
str_rindex("Some text text", "text")  // return 10
```

Function headers:

```
std::int64_t  str_index(const std::string& str, const char* const substr);
std::int64_t* str_index(const std::string* const str, \
                        const char* const substr);
std::int64_t  str_index(const std::string& str, const std::string& substr);
std::int64_t* str_index(const std::string* const str, \
                        const std::string& substr);
std::int64_t  str_rindex(const std::string& str, const char* const substr);
std::int64_t* str_rindex(const std::string* const str, \
                         const char* const substr);
std::int64_t  str_rindex(const std::string& str, const std::string& substr);
std::int64_t* str_rindex(const std::string* const str, \
                         const std::string& substr);
```

6.2.7 str_join *(since EVL 2.4)*

`str_join(vector_of_strings,delimiter)`
: it returns the string of concatenated vector members, delimited by a specified delimiter.

When the vector is 'nullptr', it returns 'nullptr'.

Examples of a mapping:

```
static std::vector<std::string> x{"Here", "is", "a", "hardcoded", "vector."};

*out->x_spaced = str_join(x,' ')   // return "Here is a hardcoded vector."
*out->x_dashed = str_join(x,'-')   // return "Here-is-a-hardcoded-vector."
*out->x_longer = str_join(x,"---") // return "Here---is---a---hardcoded---vector."
```

Function headers:

```
std::string  str_join(const std::vector<std::string>& strings, \
                      const char delimiter);
std::string* str_join(const std::vector<std::string*>* strings, \
                      const char delimiter);
std::string  str_join(const std::vector<std::string>& strings, \
                      const std::string_view delimiter);
std::string* str_join(const std::vector<std::string*>* strings, \
                      const std::string_view delimiter);
```

6.2.8 str_mask_left, str_mask_right *(since EVL 2.1)*

Functions return string with visible characters replaced by given character from given direction, but keep the specified number of character unchanged.

Example:

```
str_mask_left("abcd  text efgh", 6)   // returns "abcd  tex* ****"
str_mask_right("1234567890", 3, '-')  // returns "---4567890"
```

Without the second argument, asterisk '*' is assumed.

When the first argument is 'nullptr', these functions return 'nullptr'.

Function headers:

```
std::string  str_mask_left(const std::string& str, \
                 const std::size_t keep, const char ch = '*');
std::string* str_mask_left(const std::string* const str, \
                 const std::size_t keep, const char ch = '*');
std::string  str_mask_right(const std::string& str, \
                 const std::size_t keep, const char ch = '*');
std::string* str_mask_right(const std::string* const str, \
                 const std::size_t keep, const char ch = '*');
```

6.2.9 str_pad_left, str_pad_right *(since EVL 2.1)*

Add from left/right the specified character (space by default), up to the given length. It counts Bytes, not characters, so be careful with multibyte encodings.

Example:

```
str_pad_left("123",7,'0')     // returns "0000123"
str_pad_right("text",7)       // returns "text   "
str_pad_right("text",2)       // returns "text"
str_pad_left("Groß",6,'*')    // returns "*Groß" as "ß" has 2 Bytes
```

When the first argument is '`nullptr`', these functions return '`nullptr`'.

Function headers:

```
std::string  str_pad_left(const std::string& str, \
                 const std::size_t length, const char ch = ' ');
std::string* str_pad_left(const std::string* const str, \
                 const std::size_t length, const char ch = ' ');
std::string  str_pad_right(const std::string& str, \
                 const std::size_t length, const char ch = ' ');
std::string* str_pad_right(const std::string* const str, \
                 const std::size_t length, const char ch = ' ');
```

6.2.10 str_replace *(since EVL 1.3)*

Examples:

```
str_replace("Some text", ' ', '-')        // returns "Some-text"
str_replace("Some text", "Some", "Any")   // returns "Any text"
str_replace("Some text", ' ', "SPACE")    // returns "SomeSPACEtext"
```

When the first argument is '`nullptr`', it returns '`nullptr`'.

In mapping it might look like this:

```
out->name = str_replace(in->name", ' ', '-');
```

Function headers:

```
std::string  str_replace(const std::string& str, \
                 const char old_ch, const char new_ch);
std::string* str_replace(const std::string* const str, \
                 const char old_ch, const char new_ch);
std::string  str_replace(const std::string& str, \
                 const char* const old_substr, const char* const new_substr);
std::string* str_replace(const std::string* const str, \
                 const char* const old_substr, const char* const new_substr);
std::string  str_replace(const std::string& str, \
                 const std::string& old_substr, const std::string& new_substr);
std::string* str_replace(const std::string* const str, \
                 const std::string& old_substr, const std::string& new_substr);
```

6.2.11 str_to_base64, base64_to_str *(since EVL 2.6)*

Encode/decode string to/from Base64 form.

When the first argument is '`nullptr`', it returns also '`nullptr`'.

Examples:

```
str_to_base64("Some\r\nbíňářý text.")   // return "U29tZQ0KYsOtxYjDocWZw70gdGV4dC4="
base64_to_str("U29tZQ0KYsOtxYjDocWZw70gdGV4dC4=")   // return "Some\r\nbíňářý text."
```

Function headers:

```
std::string  str_to_base64(const std::string& str);
std::string* str_to_base64(const std::string* const str);
std::string  base64_to_str(const std::string& str);
std::string* base64_to_str(const std::string* const str);
```

6.2.12 str_to_hex, hex_to_str *(since EVL 2.0)*

Convert string or ustring to its hexadecimal representation and vice versa. (Ustring support has been added in EVL v2.6.)

When the first argument is '`nullptr`', it returns also '`nullptr`'.

Examples:

```
str_to_hex("Some text")            // return "536f6d652074657874"
hex_to_str("536f6d652074657874")   // return "Some text"
```

Function headers:

```
std::string  str_to_hex(const std::string& str);
std::string* str_to_hex(const std::string* const str);
ustring      str_to_hex(const __detail::u16str& str);
ustring*     str_to_hex(const ustring* const str);
std::string  hex_to_str(const std::string& str);
std::string* hex_to_str(const std::string* const str);
ustring      hex_to_str(const __detail::u16str& str);
ustring*     hex_to_str(const ustring* const str);
```

6.2.13 substr *(since EVL 2.0)*

Return a substring starting after given position with the specified length.

Example:

```
substr("123456789",0,2)      // returns "12"
substr("123456789",6)        // returns "789"
```

Without the third argument, it returns the rest of the string.

When the first argument is '`nullptr`', function returns '`nullptr`'.

Function headers:

```
std::string  substr(const std::string& str, const std::size_t pos = 0,
       const std::int64_t count = std::numeric_limits<std::int64_t>::max());
std::string* substr(const std::string* const str, const std::size_t pos = 0,
       const std::int64_t count = std::numeric_limits<std::int64_t>::max());
```

6.2.14 trim, trim_left, trim_right *(since EVL 1.0)*

Example:

```
trim("  text ")              // returns "text"
trim_left("  text ")         // returns "text "
trim_right("--text---", '-') // returns "--text"
```

Trim character '`char`' from both sides, from left, from right, respectively. Without the second argument, space is assumed.

When the first argument is '`nullptr`', these functions return '`nullptr`'.

Function headers:

```
std::string  trim(const std::string& str, const char ch = ' ');
std::string* trim(const std::string* const str, const char ch = ' ');

std::string  trim_left(const std::string& str, const char ch = ' ');
std::string* trim_left(const std::string* const str, const char ch = ' ');

std::string  trim_right(const std::string& str, const char ch = ' ');
std::string* trim_right(const std::string* const str, const char ch = ' ');
```

6.2.15 uppercase, lowercase *(since EVL 1.0)*

Examples:

```
uppercase("AbCd")   // returns "ABCD"
lowercase("AbCd")   // returns "abcd"
```

When the argument is '`nullptr`', these functions return '`nullptr`'.

Without specifying the second parameter it acts only on '`A-Z`' and '`a-z`'.

When there is a need to acts also on national letters (with diacritics for example), there can be the second parameter specified with the locale:

```
static std::locale de_locale("de_DE.utf8");
*out->field_upcase = uppercase(*in->field, de_locale);
```

It is possible to specify the locale in the function as string, but using the static specification of locale is recommended due to performance.

Function headers:

```
std::string  uppercase(const std::string& str);
std::string* uppercase(const std::string* const str);
std::string  uppercase(const std::string& str, const std::locale& locale);
std::string* uppercase(const std::string* const str, const std::locale& locale);

std::string  lowercase(const std::string& str);
std::string* lowercase(const std::string* const str);
std::string  lowercase(const std::string& str, const std::locale& locale);
std::string* lowercase(const std::string* const str, const std::locale& locale);
```

6.3 Checksum Functions

`md5sum(str)`
`sha224sum(str)`
`sha256sum(str)`
`sha384sum(str)`
`sha512sum(str)` *(since EVL 1.0)*

these standard checksum functions can be used in mapping this way for example:

```
*out->anonymized_username = sha256sum(*in->username);
```

When the argument is ‘`nullptr`’, it returns ‘`nullptr`’. But in such case you need to use pointer manipulation, so the example would look like:

```
out->anonymized_username = sha256sum(in->username);
```

Functions headers:

```
std::string  md5sum(const char* const str);
std::string  md5sum(const std::string& str);
std::string* md5sum(const std::string* const str);

std::string  sha224sum(const char* const str);
std::string  sha224sum(const std::string& str);
std::string* sha224sum(const std::string* const str);

std::string  sha256sum(const char* const str);
std::string  sha256sum(const std::string& str);
std::string* sha256sum(const std::string* const str);

std::string  sha384sum(const char* const str);
std::string  sha384sum(const std::string& str);
std::string* sha384sum(const std::string* const str);

std::string  sha512sum(const char* const str);
std::string  sha512sum(const std::string& str);
std::string* sha512sum(const std::string* const str);
```

6.4 IP Addresses Functions

Typical IPv4 manipulation usage within a mapping:

```
// convert and assign IPv4 string into unsigned integer
out->ipv4_uint = str_to_ipv4(in->ipv4_string);
// or the other way
out->ipv4_string = ipv4_to_str(in->ipv4_uint);
```

Typical IPv6 manipulation usage within a mapping:

```
// suppose in->ipv6_string = "4567::123"
out->ipv6_normalized = ipv6_normalize(in->ipv6_string);
    // return "4567:0000:0000:0000:0000:0000:0000:0123"

// suppose in->ipv6_string = "0000:0000:0000:0004:5678:9098:0000:0654"
out->ipv6_compressed = ipv6_compress(in->ipv6_string);
    // return "::4:5678:9098:0000:654"
```

Or one can distinguish both IP versions:

```
if ( is_valid_ipv4(in->ip_string) ) {
  // act on IPv4
}
else if ( is_valid_ipv6(in->ip_string) ) {
  // act on IPv6
}
else {
  // act when neither is valid
}
```

There are these two rules in all IP manipulation functions described in this section:

- When the first argument is a pointer, the function returns also a pointer.
- When the first argument is '`nullptr`', the function returns '`nullptr`' as well.

6.4.1 IPv4 Functions *(since EVL 2.4)*

'`ipv4addr`'
: constructor

'`str_to_ipv4()`'
: convert string to uint32,

'`ipv4_to_str()`'
: convert uint32 to ipv4 string,

'`is_valid_ipv4()`'
: to check whether the string is valid IPv4.

6.4.2 IPv6 Functions *(since EVL 2.4)*

'`str_to_ipv6()`'
: convert string to uint128,

'`ipv6_to_str()`'
: convert uint128 to ipv6 string,

'`is_valid_ipv6()`'
: to check whether the string is valid IPv6,

'`ipv6_normalize()`'
: convert string to normalized IPv6 string,

'`ipv6_compress()`'
: convert string to compressed IPv6 string,

Examples

To get normalized and compressed IPv6:

```
// suppose in->ipv6_string = "0000:0000:22::0003:4"
out->ipv6_normalized = ipv6_normalize(in->ipv6_string);
    // "0000:0000:0022:0000:0000:0000:0003:0004"
out->ipv6_compressed = ipv6_compress(in->ipv6_string);
    // "0:0:22::3:4"
```

7 Custom Functions

There is one example of custom anonymization function in `lib/functions.cpp` when creating sample project:

```
anonymize_rc()
```

This particular function is written in C++ and is country specific, it is a citizen ID in Czechia and Slovakia. The number need to confirm some rules, so you need to anonymize such values unusual way to keep the validity of such number.

You can use such function as an example and add your custom C++ codes into `lib/` project subdirectory.

Or once you consider that your custom function would be useful also for others, or you need help with C++ code, do not hesitate to contact us at support@evltool.com.

8 Examples

8.1 Connect to Oracle

Example of variables in `configs/anon/some_oracle_source.evl` which need to be set to connect to Oracle DB:

```
export SOURCE_DB="some_schema"
export TARGET_DB="some_schema"
export TARGET_TABLE_SUFFIX="_anon"

export ORAUSER="some_user"
export ORAPASS="$(cat $HOME/.orapass)"
export ORACONN="(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
                                                    (HOST=12.34.56.78)
                                                    (PORT=1521)))
                                (CONNECT_DATA=(SERVER=DEDICATED)
                                              (SERVICE_NAME=abcd)))"

export TARGET_CONN="12.34.56.79:1521/abcd"
```

Function Index

Variables Index

Variables contain also their default values if any.