EVL Workflow

version 2.8

support@datif.ai

This manual is for **EVL Workflow** (version 2.8), a code based orchestration tool. Workflows can be defined or generated to run *tasks* in specified order, in parallel, on specified target host, and at the same time consider defined settings, e.g. maximum tasks running in parallel, or prioritize some tasks over others, etc.

Table of Contents

1 Introduction

EVL Workflow is a code based orchestration tool. In general a workflow is one or more DAGs (Directed Acyclic Graphs) where vertices are some tasks (e.g. shell script or actually any executable in general) and edges specifies successors.

1.1 EVL workflow

EVL workflow is a text file, resolved as Bash script, which defines how to run *tasks*. They can be run:

- in given order,
- in parallel,
- on local machine or on particular target host (by ssh, on Kubernetes).

Tasks within a workflow are managed considering several conditions, e.g. by:

- defined maximum tasks running in parallel,
- specified priorities of each task,
- the history of CPU and memory consumption or by specified expected consumption,
- specified maximum run time,
- defined number of retries in case of failure.

These *EVL workflows* can be defined *manually* or can be *generated*, and can be easily tracked by version control system like *git*.

For particular EVL workflow examples, including examples how to run tasks based on various calendars, see Chapter 4 [Examples], page 7.

1.2 Task

EVL workflow run *tasks* where each task is one of the following:

shell script (`job/*.sh`)
: any shell script with `.sh` suffix, usually a wrapper to fire an ETL job

EVL job (parameter file `job/*.evl` with template `evs/*.evs`)
: it can be either full featured ETL job (then *EVL Tool* is needed) or it can also be a set of parameters for a shell script. Parameter `job/*.evl` file contains a set of variables assignments and run template `evs/*.evs` file which is interpreted as shell script. Such a template can be for example a wrapper for another ETL tool.

EVL workflow (parameter file `workflow/*.ewf` with template `ews/*.ews`)
: parameter `workflow/*.ewf` file contains a set of variables assignments and run template `ews/*.ews` file which is interpreted as shell script. Such a template contains the definition of the order of tasks, what to run in parallel etc.

Wait for a file
: to sniff for existence of a file with given file mask.

EVL workflow consists (mostly) of `Run` commands, which are used in EWS workflow structure definition file (`ews/*.ews`), and which fires tasks. For details, see Section 5.16 [Run], page 28.

1.3 Project

Scope of workload isolated from the business perspective. *EVL Workflow* supports inter-project workflow dependencies, but it is better when the logic stays in workflows within one project. Technically is the project represented by one directory and the name of the directory is the name of the project. That's why the project name must be unique for given user on given machine.

In the project directory is `project.sh` (bash) file, `crontab.sh` and directories of all other EVL configuration files:

`evs/*.evs`
: EVL job structure files, i.e. templates for EVL jobs

`ews/*.ews`
: EVL workflow structure files, i.e. templates for EVL workflows

`job//*.evl, job/*.sh`
: `*.evl` files contain parameters for EVL job structures (`evs`) files, `*.sh` files are standard shell scripts `*.sh`. Subfolders are considered as part of job names

`workflow/*.ewf`
: EVL workflow parameter files for `ews` files Subfolders are considered as part of workflow names

`crontab.sh`
: based on this file an EVL section of current user's crontab is created/updated by `evl crontab` command

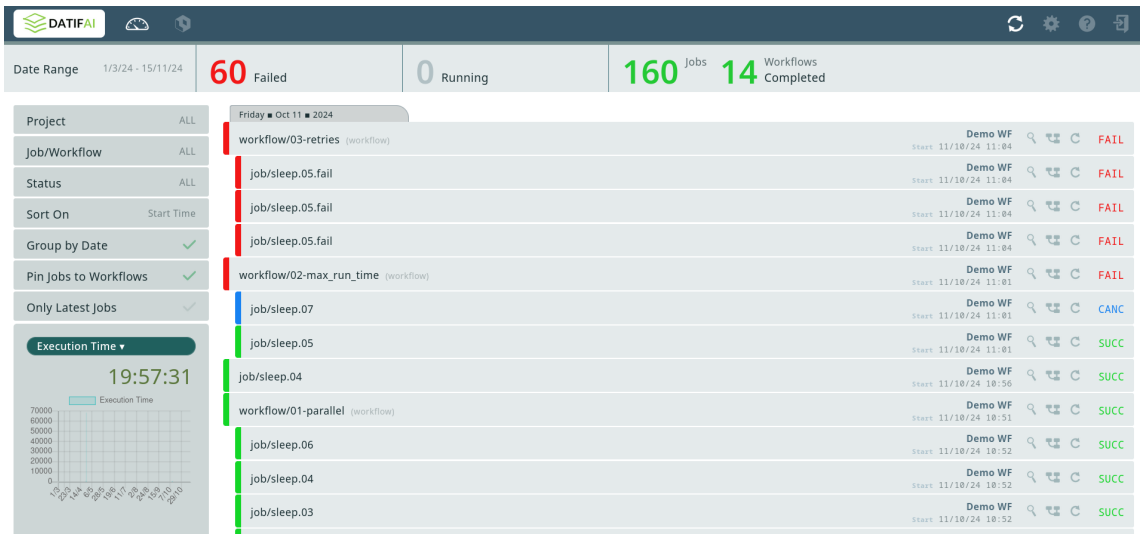
`project.sh`
: this file is interpreted as `bash` script at the beginning of each job in the project. Usually contains all project wide variables and is mandatory

> **Important:** All files `evs`, `ews`, `*.evl`, `*.ewf` are interpreted as bash scripts.

For detailed information about managing projects, see Section 5.13 [Project], page 24.

1.4 EVL Manager

EVL Workflow can be used standalone with a command line interface (CLI) or with a graphical web interface called *EVL Manager*.



EVL Manager has its own documentation.

1.5 EVL Data Hub

EVL Workflow together with *EVL Data Hub* can be used to define and schedule workflows managed by metadata. For example to:

transfer data
: either simple file copy, or also can change the file format on the way, or load data from a file to database table and vice versa,

historize data
: data can be historized as SCD2 for example

anonymize data
: data can be anonymized based on metadata annotations

generate data
: data can be generated based on metadata

EVL Data Hub has its own documentation.

2 Release Notes

Versions numbering: EVL *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

Versions 1.0–1.3 (2017/07)
: Part of the initial version of "classical" ETL tool.

Version 2.0 (2018/11)
: *EVL Job Manager* renamed to *EVL Worklow*, man pages added.

Version 2.2 (2019/10)
: Integration to *EVL Manager* – graphical web UI,

[Version 2.4], page 4, (2020/10)
: Crontab generation.
New commands: `cancel`, `crontab`, `info`, `manager`, `Sleep`.

[Version 2.8], page 4, (2023/04)
: TBA

Version 2.4

Released 2020/10

New features

- EVL Manager – EVL Microservices integrated
- Crontab generation for scheduling workflows
- Integration of secret passwords handling

New commands

- '`cancel`'
- '`crontab`'
- '`info`'
- '`manager`'
- '`Sleep`' – to have sleep command integrated

Version 2.8

Released 2023/04

3 Installation and Settings

EVL Installation

Settings after installation

3.1 Linux – RPM

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

Package name might have different name, it depends on the system and/or edition. For example for CentOS version 8 it would be:

```
sudo dnf install evl-utils-2.8.1-*.noarch.rpm
```

3.2 Linux – DEB

i.e. Ubuntu, Debian, etc.

Package name might have different name, it depends on the edition. For example for Debian it would be:

```
sudo apt-get install evl-utils_2.8.1-*_all.deb
```

3.3 Other Unix systems

i.e. Mac OS, etc.

Basically any standard Unix system with Bash and couple of standard utilities (gettext, binutils, coreutils) and libraries (boost, snappy, xml2, etc.) is possible.

Ask `support@evltool.com` for help.

3.4 Settings

EVL installation resides (usually) in

```
/opt/evl
```

To initiate EVL for current user, run

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

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

Then one can check, add or modify some 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 is ready to use. Good to start is to create new project with some sample data, jobs and workflows:

```
evl project create --sample my_first_sample
```

3.5 Text Editor

All you need to work with EVL is having a text editor.

Whatever text editor is your favourite, it is good to set syntax highlighting this way:

Syntax	**File mask**
Bash	`*.evl`, `*.evs`, `*.ewf`, `*.ews`

3.5.1 Vim

To achieve syntax to be properly highlighted in Vim, just add these lines into your `~/.vimrc` file:

```
" EVL settings "
autocmd BufRead,BufNewFile *.ev[slc] set syntax=sh
autocmd BufRead,BufNewFile *.ew[fs] set syntax=sh
autocmd BufRead,BufNewFile *.ev[md] set syntax=c
```

4 Examples

4.1 Simple example

Definition of a workflow can be defined in `ews/simple.ews` and can looks like this:

```
Run job/example.1.evl job/example.2.sh
Run job/example.3.sh
End
```

As this workflow has no parameters, the file `workflow/simple.ewf` will be empty. To fire this workflow run in command line:

```
evl run workflow/simple.ewf
```

It runs jobs `example.1.evl` and `example.3.sh` in parallel, and once `example.1.evl` finish successfully, fires also `example.2.sh`. When all jobs finish successfully, workflow is successful as well.

4.2 Example with retries and maximal run time check

Slightly advanced workflow definition might look like this:

```
Run 2h    job/common_job.sh  --project=/full/path/to/other/project \
    4h 2r job/stage.invoices.evl
End
```

which means to run `common_job.sh` from other project and kill the job and fail if not finished within 2 hours.

If `common_job.sh` finish successfully then run `job/stage.invoices.evl`. Fail if (each run) does not finish within 4 hours, but try to restart two times in case of failure.

4.3 Examples with waiting for file to exist

Following workflow definition says to wait at most 1 day for any file of the mask `/data/landing/invoice.????????.csv` on local machine. Once such file appears `job/stage.invoices.evl` is fired. When no such file exist even after 24 hours, workflow fails.

```
Run 1d w /data/landing/invoice.????????.csv \
    job/stage.invoices.evl
End
```

The same example but waiting for a file to be delivered on remote location, here on AWS S3:

```
export AWS_PROFILE="landing_zone"
Run 1d w s3://some_bucket/data/landing/invoice.????????.csv \
    job/stage.invoices.evl
End
```

The same example but waiting for a file to be delivered on remote machine, here connected over sftp:

```
EVL_RUN_SSH_OPTS="-i $HOME/.ssh/id_rsa"

Run 1d w sftp://some_user@some_host:22/data/landing/invoice.????????.csv \
    job/stage.invoices.evl
End
```

And the same '`sftp`' example, but shifting the regular checking of existence of the file to the remote machine:

```
EVL_RUN_SSH_USER="some_user"
```

```
EVL_RUN_SSH_HOST="some_host"
EVL_RUN_SSH_PORT=22
EVL_RUN_SSH_OPTS="-i $HOME/.ssh/id_rsa"

Run --target ssh 1d w /data/landing/invoice.????????.csv
Wait
Run job/stage.invoices.evl
End
```

But in this case remote monitor database has to be set up. To run the “job/stage.invoices.evl“ still on local machine, separate “Run‘ command has to be used.

5 Commands

EVL jobs are defined in `evs` files and consist of components connected by flows (pipes). But there are also several commands which (mostly) follow standard Unix commands or bash build-in functions, but operates according to URI, like '`hdfs://`' for Hadoop, '`gs://`' for Google Storage, '`s3://`' for Amazon S3 storage or '`sftp://`' for files to be accessed over SSH.

These commands can be used either in EVL jobs or in EVL workflows and also as a command line utilities.

File and directory manipulation

(Behaves by URI, i.e. '`hdfs://`', '`gs://`', '`s3://`', '`sftp://`')

EVL job and workflow specific commands

EVL task manipulation commands

Other commands

5.1 Calendar *(since EVL 2.8)*

Based on specified calendar file(s) EVL Calendar command either:

- continue processing of EVL job or workflow, or
- successfully end the task.

It compares value of '`EVL_ODATE`' (Order Date) with dates in calendar file(s).

Calendar file is simply a text file with a list of dates in format '`YYYY-MM-DD`', one date per line.

When more than one calendar is specified, they are concatenated, so behaves as logical OR.

To achive logical AND simply user several calendar commands.

Unless '`--blacklist`' option is used, whitelist of dates is assumed.

There are these predefined calendar files:

- '`first_day_of_month.cal`'
- '`last_day_of_month.cal`'
- '`workday.cal`' - i.e. Monday to Friday
- '`weekend.cal`'
- '`monday.cal`'
- '`tuesday.cal`'
- '`wednesday.cal`'
- '`thursday.cal`'
- '`friday.cal`'
- '`saturday.cal`'
- '`sunday.cal`'

These calendar files can be specified with no path, just a name. You can find them usually in '`/opt/evl/share/templates/calendar/`'

Unless '`--project`' option is used, it search in current project directory in '`calendar`' subdirectory. Then it search for common calendars (usually) in '`/opt/evl/share/templates/calendar/`'.

`Calendar`
: is to be used in EVS job structure definition file or in EWS workflow structure definition.

`evl calendar`
: is intended for standalone usage, i.e. to be invoked from command line. In this case when the '`EVL_ODATE`' do not match, the command exit with error code 1.

Synopsis

```
Calendar
  <calendar>.cal [<calendar2.cal>...]
  [--blacklist] [-p|--project=<project_dir>]
  [--no-end]

evl calendar
  <calendar>.cal [<calendar2.cal>]
  [--blacklist] [-p|--project=<project_dir>]
  [-v|--verbose]
```

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

Options

`--blacklist`
: to blacklist dates from calendar file(s)

`--no-end`
: do not end a workflow in case of no calendar match. In such case can be used in condition. See examples.

`-p, --project=<project_dir>`
: specify project folder to search for calendar file(s) in '`calendar`' subfolder

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

Examples

1. To run a workflow every working day in Czechia, put this at the beginning of EWS file:

   ```
   Calendar workday.cal
   Calendar --blacklist czech_holidays.cal
   ```

   where '`calendar/czech_holidays.cal`' supposed to be in the same project as the EWS file.
2. To run an EVL job only on Mondays, Wednesdays, and Fridays, add this line at the beginning of EVS job structure file:

   ```
   Calendar monday.cal wednesday.cal friday.cal
   ```
3. To run an EVL job only on last day of a month, but then continue processing a workflow:

   ```
   if Calendar --no-end last_day_of_month.cal
   then
     Run job/last_day_of_month.evl
   fi
   Run job/as_usual.evl
   ```

5.2 Cancel *(since EVL 2.4)*

To cancel running EVL task (i.e. job, workflow or script, or waiting for a file). Either by Run ID or by a task name and Order Date.

When there are several task(s) with given name, it tries to cancel the latest one.

It recognize type of task based on the file suffix.

`*.evl`
: EVL job

`*.ewf`
: EVL workflow

`*.sh`
: bash script

`Any other or no suffix`
: file mask of file(s) to waiting for

Synopsis

```
evl cancel
  ( <run_id>... | <task_name>... [-o|--odate=<odate>] )
  [-p|--project=<project_dir>] [-v|--verbose]

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

Options

`-o, --odate=<odate>`
: to specify particular Order Date, environment variable '`EVL_ODATE`' is then ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

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

Examples

1. To cancel a job with Run ID 145:
   ```
   evl cancel 145
   ```
2. To cancel jobs of Run IDs between 145 and 150:
   ```
   evl cancel {145..150}
   ```
3. To cancel a workflow '`billing.ewf`' of project '`/data/project/billing`' with yesterday Order Date:
   ```
   evl cancel --odate=yesterday billing.ewf --project=/data/project/billing/
   ```

5.3 Cp

(since EVL 1.3)

Copy `<source>` to `<dest>`, or multiple `<source>`(s) to `<dest>` directory. `<source>` and `<dest>` might be one of:

```
<local_path>
```

```
gdrive://<path>
gs://<bucket>/<path>
hdfs://<path>
s3://<bucket>/<path>
sftp://<path>
smb://<path>
```

On local files it calls standard '`cp`' command to copy files, but when `<source>` and/or `<dest>` contain URI '`hdfs://`' (i.e. Hadoop FS), then it calls appropriate commands to copy from/to such source/destination.

For example when argument starts with '`s3://`', then it is supposed to be on S3 file system and calls the function '`evl_s3_cp`', which is by default '`aws s3 cp`'.

So far it can copy to/from local path from/to some specific one or within one URI type. So not yet possible to copy dirrectly for example from S3 to G-Drive.

When more than one `<source>` is specified, then URI prefix must be the same for all of them.

`Cp`
: is to be used in EVS job structure definition file or in EWS workflow structure definition.

`evl cp`
: is intended for standalone usage, i.e. to be invoked from command line.

Using this command might keep your code clean, but for more complex copying better use appropriate commands directly, as it gives you all the available options for particular storage or protocol type.

Synopsis

```
Cp
  [-f|--force] [-p] <source>... <dest>

evl cp
  [-f|--force] [-p] <source>... <dest>
  [--verbose]

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

Options

`-f, --force`
: destination will be overwritten if exists

`-p`
: 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

Examples

These lines in EVL job (an '`evs`' file):

```
Cp hdfs:///some/path/to/file /some/local/path/
Cp /some/local/path/file hdfs:///some/path/
Cp hdfs:///some/path/file hdfs:///other/path/
Cp /some/local/path/file /other/local/path/
```

will call (with handling fails in proper EVL way):

```
evl_hdfs_get hdfs:///some/path/file /some/local/path/
evl_hdfs_put /some/local/path/file hdfs:///some/path/
evl_hdfs_cp hdfs:///some/path/file hdfs:///other/path/
cp /some/local/path/file /other/local/path/
```

where defaults for '`evl_hdfs_*`' variables are:

```
function evl_hdfs_get { hdfs dfs -get ; }
function evl_hdfs_cp  { hdfs dfs -cp  ; }
function evl_hdfs_put { hdfs dfs -put ; }
```

5.4 Chmod *(since EVL 2.3)*

Change file mode bits.

Each `<file>` is of the form

[<scheme>://][[<user>@@]<host>[:<port>]]<path> ...

For scheme '`hdfs://`' it calls function '`evl_hdfs_chmod`', which is by default '`hdfs dfs -chmod`'.

For scheme '`sftp://`' it calls function '`evl_sftp_chmod`'.

Synopsis

```
Chmod
  ( <mode>[,<mode>]... | <octal-mode> ) <file>...
  [-R|--recursive]

evl chmod
  ( <mode>[,<mode>]... | <octal-mode> ) <file>...
  [-R|--recursive] [--verbose]

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

Options

`-R, --recursive`
: change files and directories recursively

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

Examples

1. Simple usage examples:

```
Chmod hdfs:///some/path/
Chmod /some/local/machine/path/
```

2. Depends on environment, e.g. '`PROD`'/'`TEST`'/'`DEV`', might be useful to be used this way:

```
# on DEV:
OUTPUT_DIR=/data/output
# on PROD:
OUTPUT_DIR=hdfs:///data/output
```

   and then in '`evs`' file:

```
Chmod -p "$OUTPUT_DIR"
```

5.5 Crontab *(since EVL 2.4)*

Create/update/remove EVL section of given EVL project of current user's crontab based on '`crontab.sh`' file from current directory or from `<project_dir>`.

Synopsis

```
evl crontab
  ( set | get | remove )
  [-p|--project=<project_dir>] [-v|--verbose]

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

Options

`-p, --project=<project_dir>`
: specify project folder if not the current working one

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

Examples

1. There is a '`crontab.sh`' file in the EVL project created by '`evl project new`' or '`evl project sample`'. It is usually a good start.
2. Example of '`crontab.sh`' file:

```
# crontab.sh -- schedule EVL project workflows
#
# This file is processed by 'evl crontab set' command to create/update crontab entr
#

# Schedule workflows per environment
case "$EVL_ENV" in
  DEV)
    # Run workflow/sample.ewf Mo-Fr at 8:10, 10:10, 12:10, ..., 16:10
    Schedule 10 8-16/2 * * 1-5 sample.ewf --odate yesterday
  ;;
  TEST)
    # Run workflow/sample.ewf Mo-Fr at 5:10am
    Schedule 10 5 * * 1-5 sample.ewf
    # Restart (with the same Order Date) Mo-Fr at 6:10am
    Schedule --restart 10 6 * * 1-5 sample.ewf --odate yesterday
  ;;
  PROD)
    # Run workflow/sample.ewf daily at 5:10am
    Schedule 10 5 * * * sample.ewf --odate yesterday
    # Restart (with the same Order Date) at 6:10am
    Schedule --restart 10 6 * * * sample.ewf --odate yesterday
  ;;
esac
```

5.6 End

(since EVL 2.0)

Finish processing the EVL job or workflow, ignoring the rest of the EVS/EWS file.

EVS is EVL job definition file, for details see evl-evs(5). EWS is EVL workflow definition file, for details see evl-ews(5).

Synopsis

```
End

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

Options

Standard options:

`--help`

print this help and exit

`--usage`

print short usage information and exit

`--version`
: print version and exit

Examples

1. EVL job (an EVS file) might end like this:

```
...
Write  FLOW_99 /path/to/file evd/sample.evd --text-output
End
Here can be any comment, notes or pieces of code...
```

2. EVL workflow (an EWS file) might end like this:

```
...
Run some_job.evl
End
Here can be any comment, notes or pieces of code...
```

5.7 Log *(since EVL 2.4)*

Get status log entries of EVL task(s) (i.e. job, workflow or script, or waiting for a file). When `<regex>` ends with '`.evl`', '`.ewf`' or '`.sh`', it looks for appropriate task type.

Synopsis

```
evl log
  <regex>... [--state=<state>] [-o|--odate=<odate_regex>]
  [-p|--project=<project_dir>] [-v|--verbose]

evl log
  <run_id>... [-p|--project=<project_dir>] [-v|--verbose]

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

Options

`-o, --odate=<odate>`
: to specify particular Order Date

`-p, --project=<project_dir>`
: specify project folder if not the current working one

`--state=<state>`
: to specify particular state, possible are '`reru`', '`wait`', '`runn`', '`fail`', '`canc`', '`skip`', '`succ`', '`arch`', '`dele`'

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

Examples

1. To get information about tasks with Run ID between 145 and 150:
   ```
   evl log {145..150}
   ```
2. To get information about a workflow '`billing.ewf`' of project '`/data/project/billing`':
   ```
   evl log billing --project=/data/project/billing/ | grep "|ewf|"
   ```
3. To get all successfully finished jobs with OrderDate in 01/2026:
   ```
   evl log ".*" --odate "202601.." --state succ
   ```

5.8 Ls

(since EVL 2.0)

List `<dest>`, which might be one of:

```
<local_path>
gdrive://<path>
gs://<bucket>/<path>
hdfs://<path>
s3://<bucket>/<path>
sftp://<path>
smb://<path>
```

So for example when argument starts with '`hdfs://`', then it is supposed to be on HDFS file system and calls the function '`evl_hdfs_ls`', which is by default '`hadoop fs -ls`'.

Or when argument starts with '`s3://`', then it is supposed to be on S3 file system and calls the function '`evl_s3_ls`', which is by default '`aws s3 ls`'.

Otherwise act as usual '`ls`' command.

`Ls`
: is to be used in EVS job structure definition file or in EWS workflow structure definition.

`evl ls`
: is intended for standalone usage, i.e. to be invoked from command line.

Synopsis

```
Ls
  <dest>...
  [--force]
  [-Q|--quote-name]
  [-r|-R|--recursive]
  [-z|--zero]

evl ls
  <dest>...
  [--force]
  [-Q|--quote-name]
  [-r|-R|--recursive]
  [-z|--zero]
  [--verbose]
```

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

Options

`--force`
: do not fail if <dest> doesn't exist

#-m, –comma-separated # produce a comma separated list of entries

`-Q, --quote-name`
: enclose entry names in double quotes

`-r, -R, --recursive`
: list subdirectories recursively

`-z, --zero`
: line delimiter is NUL, not newline

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

Examples

1. These simple examples write result on stdout:

```
Ls hdfs:///some/path/????-??-??.csv
Ls s3:///somebucketname/path/
Ls /some/local/machine/path/*
```

2. To be used to initiate a flow in EVL job:

```
INPUT_FILES=/data/input
Run   ""    INPUT  "Ls $INPUT_FILES"
Map   INPUT ...
...
```

And then, for PROD environment, input files would be defined for example:

```
INPUT_FILES=hdfs:///data/input
```

5.9 Mail *(since EVL 2.1)*

When no <email> is specified, the default (comma separated list of) recipients are taken from environment variable '`EVL_MAIL_TO`'.

When environment variable '`EVL_MAIL_SEND`' is set to 0, then no e-mails are sent by this command. Useful to be set for non-production environments.

'`Mail`' command will call command from environment variable '`EVL_MAIL`' which suppose to be standard Unix command **mail** with possible other parameters.

```
Mail
```

is to be used either in EVL Job or in EVL Workflow structure definition file, i.e. in 'evs' or 'ews' file.

`evl mail`

is intended for standalone usage, i.e. to be invoked from command line.

EVS is EVL job definition file and EWS is EVL Workflow definition file, for details see man evl-evs(5) or ewf-ews(5).

Synopsis

```
Mail
  <subject> <message> [ <email>[,...] ]
  [-a <attachment>]... [-c <cc_email>[,...]] [-b <bcc_email>[,...]]

evl mail
  <subject> <message> [ <email>[,...] ]
  [-a <attachment>]... [-c <cc_email>[,...]] [-b <bcc_email>[,...]]

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

Options

`-a <attachment>`

attach the given file to the message, can be used several times to add more files

`-b <bcc_email>[,...]`

send blind carbon copies

`-c <cc_email>[,...]`

send carbon copies

Standard options:

`--help`

print this help and exit

`--usage`

print short usage information and exit

`--version`

print version and exit

Examples

1. Following setting will e-mail carbon copy of every e-mail to 'admin@server':

   ```
   export EVL_MAIL='mail -c "admin@server"'
   ```

   Following invocation of "Mail":

   ```
   Mail "Job Failed" "Job extract_file_billing failed."
   ```

   will actually run:

   ```
   echo "Job extract_file_billing failed." | \
       mail -c "admin@server" -s "Job Failed" "$EVL_MAIL_TO"
   ```

   and log appropriate information into EVL or EWF log.

2. For non-production environment it worth to set:

```
export EVL_MAIL_SEND=0
```

so then in example 1. you will obtain only such a warning in a log file:

```
EVL_MAIL_SEND is set to 0, so no e-mail was sent to \
   "$EVL_MAIL_TO" with subject "Job Failed".
```

5.10 Manager *(since EVL 2.4)*

To manipulate monitor database of given `<project>` used by EVL Manager.

`update`
: to update EVL Manager database based on EVL status log. When no project is specified, suppose current directory as a project folder.

Synopsis

```
evl manager user
  add <username> [-p|--password=<password>] [-a|--admin] [-v|--verbose]

evl manager project
  add <project_name> [-p|--path=<project_path>] [-v|--verbose]

evl manager user-to-project
  <project_name> <username> [-v|--verbose]

evl manager update
  [-p|--project=<project_dir>] [-v|--verbose]

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

Options

`-p, --project=<project_dir>`
: specify project folder if not the current working one

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

Examples

1. To update monitor DB according to status log of project '`billing`':

```
evl manager update -p billing
```

5.11 Mkdir *(since EVL 1.0)*

Create `<directory>`, which might be one of:

```
<local_path>
hdfs://<path>
s3://<bucket>/<path>
sftp://<path>
```

With option '`--parents`' no error if directory already exists and make parent directories as needed.

Each `<directory>` is of the form

[<scheme>://][[<user>@@]<host>[:<port>]]<path> ...

For scheme '`hdfs://`' it calls function '`evl_hdfs_mkdir`', which is by default '`hdfs dfs -mkdir`'.

For scheme '`s3://`' it calls function '`evl_s3_mkdir`'.

For scheme '`sftp://`' it calls function '`evl_sftp_mkdir`'.

Synopsis

```
Mkdir
  [-p|--parents] <directory>...

evl mkdir
  [-p|--parents] <directory>...

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

Options

`-p, --parents`
: no error if existing, make parent directories as needed

Standard options:

`--help`
: print this help and exit

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

`--version`
: print version and exit

Examples

1. Simple usage examples:

```
Mkdir hdfs:///some/path/
Mkdir /some/local/machine/path/
```

2. Depends on environment, e.g. '`PROD`'/'`TEST`'/'`DEV`', might be useful to be used this way:

```
# on DEV:
OUTPUT_DIR=/data/output
# on PROD:
OUTPUT_DIR=hdfs:///data/output
```

and then in 'evs' file:

```
Mkdir -p "$OUTPUT_DIR"
```

5.12 Mv *(since EVL 1.0)*

Move <source> to <dest>, or multiple <source>(s) to <dest> directory. <source> and <dest> might be one of:

```
<local_path>
gdrive://<path>
gs://<bucket>/<path>
hdfs://<path>
s3://<bucket>/<path>
sftp://<path>
smb://<path>
```

So for example when argument starts with 'hdfs://', then it is supposed to be on HDFS file system and calls the function 'evl_hdfs_mv', which is by default 'hadoop fs -mv'.

Or when argument starts with 's3://', then it is supposed to be on S3 file system and calls the function 'evl_s3_mv', which is by default 'aws s3 mv'.

Otherwise act as usual 'mv' command.

So far it can move to/from local path from/to some specific one or within one URI type. So not yet possible to move dirrectly for example from S3 to G-Drive.

#But when <source> and/or <dest> contain URI like 'hdfs://', 's3://', #'gs://' or 'sftp://' (i.e. Hadoop FS, Amazon S3, Google Storage and SFTP), #then it calls appropriate commands to move (or copy and delete) from/to such source/destination.

When more than one <source> is specified, then URI prefix must be the same for all of them.

`Mv`
: is to be used in EVS job structure definition file or in EWS workflow structure definition.

`evl mv`
: is intended for standalone usage, i.e. to be invoked from command line.

Using this command might keep your code clean, but for more complex copying better use appropriate commands directly, as it gives you all the available options for particular storage or protocol type.

Synopsis

```
Mv
  [-f|--force] <source>... <dest>

evl mv
  [-f|--force] <source>... <dest>
  [-v|--verbose]

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

Options

`-f, --force`
: destination will be overwritten if exists

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

Examples

1. This line in EVL job (an 'evs' file):

   ```
   Mv hdfs:///some/path/to/file /some/local/path/
   ```

   will call:

   ```
   evl_hdfs_get hdfs:///some/path/file /some/local/path/
   evl_hdfs_rm hdfs:///some/path/file
   ```

2. This line in EVL job (an 'evs' file):

   ```
   Mv /some/local/path/file hdfs:///some/path/
   ```

   will call:

   ```
   evl_hdfs_put /some/local/path/file hdfs:///some/path/
   rm /some/local/path/file
   ```

3. This line in EVL job (an 'evs' file):

   ```
   Mv hdfs:///some/path/file hdfs:///other/path/
   ```

   will call:

   ```
   evl_hdfs_cp hdfs:///some/path/file hdfs:///other/path/
   evl_hdfs_rm hdfs:///some/path/file
   ```

4. This line in EVL job (an 'evs' file):

   ```
   Mv /some/local/path/file /other/local/path/
   ```

   will call:

   ```
   mv /some/local/path/file /other/local/path/
   ```

Where defauls for '`evl_hdfs_*`' variables are:

```
function evl_hdfs_cp  { hdfs dfs -cp  ; }
function evl_hdfs_get { hdfs dfs -get ; }
function evl_hdfs_put { hdfs dfs -put ; }
function evl_hdfs_rm  { hdfs dfs -rm ; }
```

5.13 Project *(since EVL 1.0)*

Create new EVL project(s) or get project settings. Consider current directory as a project one, unless `<project_dir>` is specified with either full or relative path. Last folder in the `<project_dir>` path is considered as project name. Prefer to use small letters for project names, however numbers, capital letters, underscores and dashes are possible.

Projects can be included into another projects. But remember that parent's project.sh is not automatically included (i.e. sourced) by subproject's one.

`create <project_name> [<project_name_2>...]`
: create <project_name> directory (directories) with standard subfolders structure and default '`project.sh`' configuration file.

`create --sample <project_name> [<project_name_2>...]`
: create <project_name> directory (directories) with sample data and sample jobs and workflows.

`get <variable_name> [--path] [--omit-newline] [--project=<project_dir>]`
: get the value of <variable_name>, based on the project.sh configuration file. Search '`project.sh`' in the current directory, unless <project_dir> if mentioned. With option '`--path`', it returns path in a clean way (i.e. no multiple slashes, no slash at the end, no '`/./`', no spaces or tabs at the end or beginning). With option '`--omit-newline`', return value without trailing newline.

To drop the whole project simply delete the folder recursively.

Synopsis

```
evl project create
  <project_name>... [--sample]
  [-v|--verbose]

evl project get
  <variable_name>
  [-p|--project=<project_dir>]
  [--path] [--omit-newline]
  [-v|--verbose]

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

Options

`--omit-newline`
: return value without trailing newline, good for example for assigning returned value into a variable

`--path`
: it returns path in a clean way (i.e. no multiple slashes, no slash at the end, no '`/./`', no spaces or tabs at the end or beginning)

`-p, --project=<project_dir>`
: specify project folder if not the current working one

`--sample`
: create project with sample configuration

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

These variables can be set in the project's '`project.sh`' file. Here are mentioned with their default values.

`EVL_PROJECT_LOG_DIR="$EVL_LOG_PATH/<project_name>"`
: where all the project logs goes

`EVL_PROJECT_TMP_DIR="$EVL_TMP_PATH/<project_name>"`
: where all the temporary files should be placed for all tasks of the project

Examples

1. To create three main projects with couple of subprojects:

```
# shared to all projects
evl project create shared

evl project create stage       # shared stuff only for "stage" projects
evl project create stage/sap stage/tap stage/erp stage/signaling

evl project create dwh         # shared stuff only for "dwh" projects
evl project create dwh/usage dwh/billing dwh/party dwh/contract dwh/product

evl project create mart        # shared stuff only for "mart" projects
evl project create mart/marketing mart/sales
```

2. To create new project with sample data, jobs and workflows:

```
evl project create --sample my_sample
```

3. To get the project path to log directory (i.e. '`EVL_PROJECT_LOG_DIR`'):

```
evl project get --path EVL_PROJECT_LOG_DIR
```

5.14 Rm

(since EVL 2.0)

Remove `<dest>`, which might be one of:

```
<local_path>
hdfs://<path>
gs://<bucket>/<path>
s3://<bucket>/<path>
sftp://<path>
```

So for example when argument starts with '`hdfs://`', then it is supposed to be on HDSF file system and calls the function '`evl_hdfs_rm`', which is by default '`hadoop fs -rm`'.

Or when argument starts with '`s3://`', then it is supposed to be on S3 file system and calls the function '`evl_s3_rm`', which is by default '`aws s3 rm`'.

In all other cases standard '`rm`' command is used.

Synopsis

```
Rm
  [-f|--force] [-r|-R|--recursive] <dest>...

evl rm
```

```
    [-f|--force] [-r|-R|--recursive] <dest>...
    [--verbose]

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

Options

`-f, --force`
: ignore nonexistent files and arguments, never prompt

`-r, -R, --recursive`
: remove directories and their contents recursively

Examples

1. To remove file from HDFS:

```
Rm hdfs://some/path/to/file
```

5.15 Rmdir

(since EVL 2.6)

Remove `<directory>` on local filesystem or on HDFS in case `<directory>` starts with '`hdfs://`' or on remote machine by '`ssh`' when starts with '`sftp://`'. It fails if the directories are empty.

Each `<directory>` is of the form

[<scheme>://][[<user>@@]<host>[:<port>]]<path> ...

For scheme '`hdfs://`' it calls function '`evl_hdfs_rmdir`', which is by default '`hdfs dfs -rmdir`'.

For scheme '`sftp://`' it calls function '`evl_sftp_rmdir`'.

Synopsis

```
Rmdir
  [-p|--parents] <directory>...

evl rmdir
  [-p|--parents] <directory>...

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

Options

`-p, --parents`
: remove `<directory>` and its ancestors, e.g. 'Rmdir -p a/b/c' is similar to 'Rmdir a/b/c a/b a'

Standard options:

`--help`
: print this help and exit

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

`--version`
: print version and exit

Examples

1. Simple usage examples:

```
Rmdir hdfs:///some/path/
Rmdir /some/local/machine/path/
```

2. Depends on environment, e.g. '`PROD`'/'`TEST`'/'`DEV`', might be useful to be used this way:

```
# on DEV:
OUTPUT_DIR=/data/output
# on PROD:
OUTPUT_DIR=hdfs:///data/output
```

   and then use in '`evs`' file:

```
Rmdir -p "$OUTPUT_DIR"
```

   and do not care if you operate locally or on HDFS.

5.16 Run *(since EVL 1.0)*

EVL Run command runs EVL task(s), i.e. `<job>`, `<workflow>`, or any shell `<script>`, or wait for a file to exist.

Tasks are provided as list of arguments or in a `<file.csv>` with '`--from-file`' option.

`Run`
: run EVL task(s) from EVL workflow (i.e. from an '`ews/*.ews`' file)

`evl run`
: is intended for standalone usage, i.e. to be invoked from command line

Type of the task is recognized by a file suffix:

`*.evl`
: suppose an EVL job, either full path or relative to project directory or relative to project's '`job/`' subfolder

`*.ewf`
: suppose an EVL workflow, either full path or relative to project directory or relative to project's '`workflow/`' subfolder

`*.sh`
: suppose a Bash script, either full path or relative to project directory or relative to project's '`job/`' subfolder

`w <file_mask>`
: in case of waiting for a file presence, wait flag '`w`' must be used followed by a file mask

If more than one task is provided, then run them in serial one after another, i.e. run the following one after previous successfully finished. In case of option '`--dependencies-file`' the order is taken from provided dependencies CSV file.

Failures and retries:

Once one EVL task fails, then whole '`Run`' or '`evl run`' command fails (unless '`$EVL_RUN_FAIL`' environment variable is set to 0, or option '`--ignore-fail`' is specified, but use with care).

In case of EVL task failure, '`Run`' or '`evl run`' command tries to restart it automatically '`$EVL_RUN_RETRY`' times. If `<retries>` is specified, then such value has precedence over '`$EVL_RUN_RETRY`'.

By option '`--run-on-fail`' a task to be run can be specified in case of a failure.

Maximal run time:

When the run time of given EVL task exceed '`$EVL_RUN_TIME`', then such task is killed and '`Run`' or '`evl run`' command fails. If `<time>` is specified, then such value has precedence over '`$EVL_RUN_TIME`'. Each retry measure run time from the beginning.

`<time>` can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

Synopsis

```
Run
  ( [<time>[smhd]] [<retries>r] ( <job> | <workflow> | <script> | w <file_mask> ) )...
  [ --target k8s|local|ssh ]
  [ --ignore-fail ]
  [ --run-on-fail=<task> ]
  [ --dependencies-file=<file.csv> ]
  [-c|--check-prev-run]
  [-D|--define=<definition>]...
  [-o|--odate=<yyyymmdd>]
  [-p|--project=<project_dir>]
  [-r|--restart]

Run
  --from-file=<file.csv>
  [ --dependencies-file=<file.csv> ]
  [-c|--check-prev-run]
  [-D|--define=<definition>]...
  [-o|--odate=<yyyymmdd>]
  [-p|--project=<project_dir>]
  [-r|--restart]

evl run
  ( [<time>[smhd]] [<retries>r] ( <job> | <workflow> | <script> | w <file_mask>) )...
  [ --target k8s|local|ssh ]
  [ --ignore-fail ]
  [ --run-on-fail=<task> ]
  [ --dependencies-file=<file.csv> ]
  [-c|--check-prev-run]
  [-D|--define=<definition>]...
  [-o|--odate=<yyyymmdd>]
  [-p|--project=<project_dir>]
  [-r|--restart]
  [-s|--progress]
  [-v|--verbose]
  [ --monitor-db=<monitor_db_uri> ]
  [ --parent-run-id=<parent_run_id> ]

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

Options

`-c, --check-prev-run`
: check if given task(s) already finished for given Order Date and fail with exit code 2 if yes

`-D, --define=<definition>`
: the `<definition>` is evaluated right before running a task, but after evaluating settings from '`evl`' or '`ewf`' file, e.g. '`-DSOME_PATH=/some/path`' will do '`eval SOME_PATH=/some/path`', and overwrites then variable SOME_PATH possibly defined in '`evl`' or '`ewf`' file. Multiple '`--define`' options can be used.

`--dependencies-file=<file.csv>`
: read arguments from a CSV `<file.csv>`. CSV file (delimited by $EVL_CONFIG_FIELD_SEPARATOR, which is by default '`,`') must have a header and is of the form:

```
task_name;dependency_task_name;comment;rest_is_ignored
```

where:

`task_name`
: is path to a task relative to the project directory. In case of wait-for-file, it can be any file mask.

`dependency_task_name`
: is path to a task relative to the project directory. In case of wait-for-file, it can be any file mask.

`comment and the rest of the line`
: whatever for documentation purposes, is ignored. But no newlines!

`--from-file=<file.csv>`
: read arguments from a CSV `<file.csv>`. CSV file (delimited by $EVL_CONFIG_FIELD_SEPARATOR, which is by default '`,`') must have a header and is of the form:

```
task_name;max_time;retries;wait_for_file;target_type;nice;
    ignore_fail;run_task_on_fail;valid_from;valid_to;comment;rest_is_ignored
```

where:

`task_name`
: is path to a task relative to the project directory. In case of wait-for-file, it can be any file mask. All (non wait-for-file) tasks must exist on filesystem, otherwise workflow fail.

`max_time`
: is maximal run time, see above

`retries`
: is the number of retries, see above

`wait_for_file`
: is '`Yes`'/'`No`' flag if the task_name is to be executed or is a file mask to wait for. If empty, '`No`' is assumed.

`target_type`
: possible values are '`local`' (default), '`ssh`' or '`k8s`'.

`nice`
: run task with this nice value, which is a number between 0 and 19. It is the standard Linux Nice value.

`ignore_fail`
: is '`Yes`'/'`No`' flag if failure of the task '`task_name`' can be ignored or not. Default is '`No`'.

`run_task_on_fail`
: run this task in case of failure of task '`task_name`'.

`valid_from`
: run task only if ODATE is greated than or equal to this value of the format YYYY-MM-DD. If empty, consider task valid.

`valid_to`
: run task only if ODATE is less than or equal to this value of the format YYYY-MM-DD. If empty, consider task valid.

`comment and the rest of the line`
: whatever for documentation purposes, is ignored. But no newlines!

`--ignore-fail`
: by default once some task to be run fails, the whole workflow fails (after other '`Run`' commands finish and reach first '`Wait`'). This option ignores this and allows other tasks specified by given '`Run`' command to finish. Also the whole workflow continues.

`--monitor-db=<monitor_db_uri>`
: specify Postgres DB to be used for monitoring

`-o, --odate=<yyyymmdd>`
: run evl job with specified Order Date, environment variable '`$EVL_ODATE`' is ignored

`-s, --progress`
: for an EVL job it shows the number of records passed each component, for an EVL workflow it shows the states of each component, and for running shell scripts it does nothin. The output is refreshed every '`$EVL_PROGRESS_REFRESH_SEC`' seconds. By default it is 2 seconds.

`-p, --project=<project_dir>`
: specify project folder if not the current working one

`--parent-run-id`
: for monitoring purpose in case of non-local targets, specifies under which workflow are EVL tasks invoked

`-r, --restart`
: do not continue given workflow(s), but restart them from the beginning

`--run-on-fail=<task>`
: when some EVL task fails, run this `<task>`. When used together with '`--ignore-fail`', then this '`run-on-fail <task>`' is fired immediately after the task fails and then continue with others. If also some other task fails, then this '`run-on-fail <task>`' is fired again.

`--target=( k8s | local | ssh )`
: run tasks on particular target, possible values are:

  `k8s`
  : run on Kubernetes cluster which is defined by '`$EVL_RUN_K8S_*`' variables

  `local`
  : run on local machine. This is the default value.

`ssh`

run on a remote machine connected over ssh defined by '`$EVL_RUN_SSH_*`' variables

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 variables which controls EVL Workflow '`Run`' or '`evl run`' command behaviour. With their default values. These variables can be set for example in user's '`~/.evlrc`' file or in the project's '`project.sh`'.

Control failures:

`EVL_RUN_FAIL=1`

whether or not to fail given '`Run`' command once any EVL task fails, so when zero is set, the '`Run`' command continue regardless tasks failures

`EVL_RUN_FAIL_MAIL=1`

whether or not to send an e-mail when the task fails

`EVL_RUN_FAIL_MAIL_SUBJECT='$EVL_PROJECT FAILED'`

subject of such e-mail, where variables are resolved by '`envsubst`' utility in time of failure

`EVL_RUN_FAIL_MAIL_MESSAGE`

message of such e-mail, by default it is:

```
Project:     $EVL_PROJECT
Workflow:    $EVL_WORKFLOW
Task:        $EVL_TASK
Order Date:  $EVL_ODATE
Sent to:     $EVL_MAIL_TO
Task log:    $EVL_TASK_LOG
Tail of log: $(tail $EVL_TASK_LOG)
```

where commands '`$(...)`' are resolved and also all variables are substituted (by '`envsubst`' utility).

`EVL_RUN_FAIL_SNMP=0`

whether or not to send SNMP trap when the task fails.

`EVL_RUN_FAIL_SNMP_MESSAGE='$EVL_PROJECT FAILED'`

SNMP message to be send.

Control retries:

`EVL_RUN_RETRY=0`
: the number of times it retries to run the task again. Zero means no retry and fail '`Run`' command once the given task fails.

`EVL_RUN_RETRY_INTERVAL=5m`
: the amount of time between retries. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

Control parallel runs:

`EVL_RUN_MAX_PARALLEL=16`
: how many tasks in the workflow can run at once in parallel in one stage (i.e. before next '`Wait`')

`EVL_RUN_MAX_PARALLEL_CHECK_SEC=10`
: how many seconds to wait between checking maximum parallel runs

Control waiting:

`EVL_RUN_DEPENDENCIES_CHECK_SEC=10`
: how many seconds to wait between checking dependencies from dependencies file.

`EVL_RUN_TIME=24h`
: maximal run time, so if the task invoked by '`Run`' command is not finished after this amount of time, it is killed. The time is counted since the task is really running, not since the invocation (i.e. waiting time is not included). It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_FILE_INTERVAL=5m`
: the time interval between each check for a file(mask) existence. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_FILE_TIME=10h`
: maximal amount of time to wait for a file(mask). It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_LOCK=1`
: whether or not to wait for a lock file, i.e. if somebody is running the same task at the moment.

`EVL_RUN_WAIT_FOR_LOCK_INTERVAL=5m`
: the time interval between each check. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_LOCK_TIME=10h`
: maximal amount of time to wait for a lock file. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_PREV_ODATE=0`
: whether or not to automatically wait for previous Order Date of given task. Setting to 1 might be useful when you must run daily processing strictly in right order.

`EVL_RUN_WAIT_FOR_PREV_ODATE_INTERVAL=5m`
: the time interval between each check. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_RUN_WAIT_FOR_PREV_ODATE_TIME=10h`
: maximal amount of time to wait for previous Order Date. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

Warning control:

`EVL_RUN_WARN_MAIL=0`
: whether or not to send an email when there is warning

`EVL_RUN_WARN_MAIL_SUBJECT='$EVL_PROJECT WARNING'`
: subject of such e-mail, where variables are resolved by '`envsubst`' utility in time of failure

`EVL_RUN_WARN_MAIL_MESSAGE`
: message of such e-mail, by default it is:

```
Project:     $EVL_PROJECT
Workflow:    $EVL_WORKFLOW
Task:        $EVL_TASK
Order Date:  $EVL_ODATE
Sent to:     $EVL_MAIL_TO
Task log:    $EVL_TASK_LOG
Tail of log: $(tail $EVL_TASK_LOG)
```

: where commands '`$(...)`' are resolved and also all variables are substituted (by '`envsubst`' utility).

`EVL_RUN_WARN_SNMP=0`
: whether or not to send SNMP trap when there is a warning

`EVL_RUN_WARN_SNMP_MESSAGE='$EVL_PROJECT WARNING'`
: SNMP message to be send in such case

Kubernetes control:

`EVL_RUN_K8S_CONTAINER_IMAGE="evl-tool:latest"`
: an image to run the task on

`EVL_RUN_K8S_CONTAINER_LIMIT_CPU="8000m"`
: maximum CPUs available for the task, decimal number is allowed, or "millicpu" can be used. E.g. "0.5" and "500m" are the same and means half of the CPU.

`EVL_RUN_K8S_CONTAINER_LIMIT_MEMORY="4Gi"`
: maximum memory for the task, it can be in Bytes, or with usual suffixes like "Gi", "Mi", "Ki" or "G", "M", "K"

`EVL_RUN_K8S_CONTAINER_LIMIT_STORAGE="40Gi"`
: maximum ephemeral storage for the task, it can be in Bytes, or with usual suffixes like "Gi", "Mi", "Ki" or "G", "M", "K"

`EVL_RUN_K8S_CONTAINER_REQUEST_CPU="2000m"`
: minimum requested CPUs for the task, decimal number is allowed, or "millicpu" can be used. E.g. "2.2" and "2200m" are the same.

`EVL_RUN_K8S_CONTAINER_REQUEST_MEMORY="1Gi"`
: minimum requested memory for the task, it can be in Bytes, or with usual suffixes like "Gi", "Mi", "Ki" or "G", "M", "K"

`EVL_RUN_K8S_CONTAINER_REQUEST_STORAGE="20Gi"`
: minimum ephemeral storage for the task, it can be in Bytes, or with usual suffixes like "Gi", "Mi", "Ki" or "G", "M", "K"

`EVL_RUN_K8S_NAMESPACE="default"`
: Kubernetes namespace under which tasks suppose to run

`EVL_RUN_K8S_PERSISTENT_BUCKET`
: mandatory variable with persistent (AWS S3) bucket, where: 1. the EVL license is stored. i.e. '`s3://<some_bucket>/evl_license_key`' 2. logs are collected in '`evl-log`' directory, i.e. '`s3://<some_bucket>/evl-log`' 3. EVL projects whose tasks to be run, e.g. '`s3://<some_bucket>/<some_project>`', (it copies the tasks' definitions from this location). The value must be only a bucket name, without '`s3://`'.

`EVL_RUN_K8S_SERVICE_ACCOUNT_NAME="default"`
: Kubernetes Service Account Name

`EVL_RUN_K8S_SHM_LIMIT="1Gi"`
: Shared Memory Size Limit of the Kubernetes Pod. (Shared memory is used by shared lookups.)

`EVL_RUN_K8S_RETRY=0`
: the '`backoffLimit`' parameter in Kubernetes job definition. This variable defines how many times to restart a Kubernetes job, this number of retires is on the Kubernetes level, so it is different from the variable '`EVL_RUN_RETRY`', which is on the current machine level.

`EVL_RUN_K8S_TTL_AFTER_FINISHED=10`
: set spec.ttlSecondsAfterFinished in Kubernetes Job definition, non-zero value is good to obtain logs from finished (i.e. Compleded or Failed) tasks. (TTL means 'Time to Live'.)

SSH control:

TBA

Examples

Commandline invocation:

1. To restart a workflow from the beginning:

   ```
   evl run --restart workflow/load_invoices.ewf
   ```

2. Following command runs an EVL job with yesterday ODATE showing progress

   evl run job/aggreg_invoices.evl –odate=yesterday –progress –project=/full/path/to/project

   or when current directory is a project one:

   ```
   evl run job/aggreg_invoices.evl --odate=yesterday --progress
   ```

Within EWS file usage:

1. To run an EVL job in an EVL Workflow (i.e. within an '`ews`' file), and try once more when job fails:

   ```
   Run 1r aggreg_invoices.evl
   ```

2. Following invocation means to run common_job.sh (fail if not finished within 2 hours) and then run '`job/stage.invoices.evl`', fail if (each run) does not finish within 4 hours, and try to restart two times when fail:

```
Run 2h    job/common_job.sh  --project=/full/path/to/other/project \
    4h 2r job/stage.invoices.evl
```

5.17 Set *(since EVL 2.0)*

Manually set status to EVL task (i.e. job, workflow or script, or waiting for a file). Keep in mind, that this command is not intended for standard usage. It is only for special cases when something went wrong and you need to set status manually.

Possible states are:

'`reru`'
: set state to '`RERU`', i.e. '`To rerun`'.

'`wait`'
: set state to '`WAIT`', i.e. '`Waiting`'.

'`runn`'
: set state to '`RUNN`', i.e. '`Running`'.

'`fail`'
: set state to '`FAIL`', i.e. '`Failed`'.

'`canc`'
: set state to '`CANC`', i.e. '`Cancelled`'.

'`skip`'
: set state to '`SKIP`', i.e. '`Skipped`'.

'`succ`'
: set state to '`SUCC`', i.e. '`Successful`'.

'`arch`'
: set state to '`ARCH`', i.e. '`Archived`'.

'`dele`'
: set state to '`DELE`', i.e. '`Deleted`'.

Synopsis

```
evl set
  <state> ( <run_id>... | <task_name>... [-o|--odate=<odate>] )
  [-p|--project=<project_dir>] [-v|--verbose]

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

Options

`-o, --odate=<odate>`
: to specify particular Order Date, environment variable '`EVL_ODATE`' is then ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

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

Examples

1. Mark '`some_job.evl`' as successfully finished with current date as '`$EVL_ODATE`':

```
evl set succ some_job.evl
```

5.18 Skip *(since EVL 2.0)*

Use '`skip`' to produce log entry like in the case an EVL task (i.e. job, workflow or script, or waiting for a file) would be successfully finished (in fact marked as '`SKIP`'). This is useful when other jobs or workflows are waiting for such EVL task.

It is actually an alias to '`evl set skip`' commands.

Synopsis

```
evl skip
  <state> ( <run_id>... | <task_name>... [-o|--odate=<odate>] )
  [-p|--project=<project>] [-v|--verbose]

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

Options

`-o, --odate=<odate>`
: to specify particular Order Date, environment variable '`EVL_ODATE`' is then ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

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

Examples

1. Mark '`some_job.evl`' as successfully finished with current date as Order Date:

```
evl skip some_job.evl --odate today
```

5.19 Sleep *(since EVL 2.4)*

Use '`Sleep`' to fire previously defined EVL tasks (i.e. jobs/workflow/scripts or waiting for a file) by '`Run`' command. Then wait specified amount of `<time>`.

`<time>` can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

Sleep is useful when need to run jobs/workflows in parallel, but shifted way. Or in the case you need to spread many short (but intensive) jobs in the time, so kind of serial, but let them overlap.

While a workflow is invoked by '`continue`', then already fired Sleep commands are ignored. So restarting a workflow this way will not stuck on already finished Sleeps.

`Sleep`
: is to be used in EVL workflow structure definition file, i.e. in EWS file.

`evl sleep`
: is intended for standalone usage, i.e. to be invoked from command line.

EWS is EVL workflow definition file, for details see man evl-ews(5).

Synopsis

```
Sleep
  <time>[smhd]

evl sleep
  <time>[smhd] [-v|--verbose]

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

Options

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

Examples

1. To run a job every 15 minutes, ignore failures:

```
export EVL_RUN_FAIL=0

let i=0
while (( i < 1440 ))
do
  # Using printf because of proper sorting by file name
  Run flowmon_load.$(printf "%04d" $i).evl
  Sleep 15m
let i+=15
done

Wait
```

2. To fire 100 jobs in parallel, but shifted by 20 seconds:

```
for i {1..100}
do
  Run $i.evl
  Sleep 20
done
Wait
```

5.20 Spark *(since EVL 2.0)*

In case jar file is specified (i.e. file with mask *.jar), it invokes:

```
$EVL_SPARK_SUBMIT <spark_submit_options> <jar_file> --name <name>
```

where EVL_SPARK_SUBMIT is '`spark-submit`' by default.

When other than jar file is used, then it firstly build the code by '`$EVL_SPARK_BUILD`', which is by default '`sbt`', and then run such jar file in above manner.

Synopsis

```
Spark
  ( <jar_file> | <scala_source> ) [--name <name>]

evl spark
  ( <jar_file> | <scala_source> ) [--name <name>]
  [--verbose]

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

Options

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

Examples

Run already built scala code in YARN:

```
export EVL_SPARK_SUBMIT="--master yarn --executor-memory 2G
                         --conf spark.executor.memoryOverhead=4G"
Spark agregate_something.jar --name aggregate_something
```

Run scala code in YARN:

```
export EVL_SPARK_SUBMIT="--master yarn --executor-memory 2G
                         --conf spark.executor.memoryOverhead=4G"
Spark agregate_something.scala --name aggregate_something
```

5.21 Status *(since EVL 2.4)*

To get the latest state of an EVL task (i.e. job, workflow or script, or waiting for a file).

Synopsis

```
evl status
  ( <run_id> | <task_name> [-o|--odate=<odate>] )
  [-p|--project=<project>] [-v|--verbose]

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

Options

`-o, --odate=<odate>`
: to specify particular Order Date, environment variable '`EVL_ODATE`' is then ignored

`-p, --project=<project_dir>`
: specify project folder if not the current working one

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

Examples

1. To get the last state of '`some_job.evl`' with today Order Date:
   ```
   evl status some_job.evl --odate today
   ```
2. To get the last state of task with Run ID 3222 of the given project:
   ```
   evl status 3222 -p /data/project/roaming
   ```

5.22 Test *(since EVL 2.1)*

On local file system works like standard GNU/Linux '`test`' command. If `<path>` starts with '`hdfs://`', then use function '`evl_hdfs_test`', which is by default '`hadoop fs -test`'. If `<path>` starts with '`s3://`', then use function '`evl_s3_test`'.

Synopsis

```
Test
  -[defsz] <path>

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

Options

`-d`

return 0 if <path> exists and is a directory

`-e`

return 0 if <path> exists

`-f`

return 0 if <path> exists and is a regular file

`-s`

return 0 if file <path> exists and has a size greater than zero

`-z`

return 0 if file <path> is zero bytes in size, else return 1

Examples

TBA

5.23 Touch *(since EVL 2.7)*

Update the access and modification times of each <file> to the current time. Each `<file>` argument that does not exist is created empty.

Each `<file>` is of the form

[<scheme>://][[<user>@@]<host>[:<port>]]<path> ...

For scheme '`hdfs://`' it calls function '`evl_hdfs_touch`', which is by default '`hdfs dfs -touchz`'.

For scheme '`sftp://`' it calls function '`evl_sftp_touch`'.

Synopsis

```
Touch
  <file>...
```

Options

Standard options:

`--help`
: print this help and exit

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

`--version`
: print version and exit

Examples

1. Simple usage examples:

```
Touch hdfs:///some/path/
Touch /some/local/machine/path/
```

2. Depends on environment, e.g. '`PROD`'/'`TEST`'/'`DEV`', might be useful to be used this way:

```
# on DEV:
OUTPUT_DIR=/data/output
# on PROD:
OUTPUT_DIR=hdfs:///data/output
```

   and then in '`evs`' file:

```
Touch "$OUTPUT_DIR"
```

5.24 Wait *(since EVL 2.1)*

It waits for successful finish of all previous components of an EVL job (in an EVS file) or all previous '`Run`' commands in an EVL workflow (in an EWS file). So it acts similar way as standard Bash '`wait`' command.

Once all previous components/parts of the job/workflow successfully finish the job/workflow continue further.

EVS is EVL job structure definition file, for details see '`man 5 evl-evs`'.

EWS is EVL workflow structure definition file, for details see '`man 5 evl-ews`'.

EVL job (EVS file):

If any component of an EVL job fails, it immediately cancel the whole job.

There is neither argument nor option for '`Wait`' command in EVL job, i.e. in EVS file. More precisely: all arguments are ignored, so it can be used for comments.

EVL workflow (EWS file):

If any invoked EVL job/workflow (by '`Run`' command) fails, then cancel only given '`Run`' command and continue in processing others. Once all '`Run`' commands finish (either successfully or fail), then '`Wait`' command fails unless '`EVL_WAIT_FAIL`' is set to 0.

When no `<job>`, `<workflow>` or `<script>` is specified, further processing will wait `<time>` for all previously fired parts (i.e. '`Run`' commands) and fail if at least one of them fails.

When no `<time>` is specified by '`--time`' option, then wait at most '`$EVL_WAIT_TIME`', which is by default 10 hours.

`<time>` can be secified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

When <job>, <workflow> or <script> is specified, it waits for successful run of such job/workflow/script of the current project and of the current <odate>. Different <project> and/or <odate> can be specified by the particular options.

Synopsis

```
Wait
  ( <job>.evl | <workflow>.ewf | <script>.sh )
  [-p|--project=<project>] [-o|--odate=<odate> | -y|--yesterday]
  [-t|--time=<time>]

Wait
  [-m|--myself [-o|--odate=<odate> | -y|--yesterday]]
  [-t|--time=<time>]

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

Options

Standard options:

`--help`
: print this help and exit

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

`--version`
: print version and exit

Environment Variables

`EVL_WAIT_FAIL=1`
: EVL workflow only. Whether or not to fail the whole workflow when the '`Run`' command fails, so when zero is set, the workflow continue regardless task failures

`EVL_WAIT_INTERVAL=2s`
: EVL workflow only. The time interval between each check for '`Wait`' command. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

`EVL_WAIT_TIME=10h`
: EVL workflow only. Maximal amount of time to wait for a previous '`Run`' commands to finish. It can be specified in seconds, minutes, hours or days, so suffix '`s`', '`m`', '`h`' or '`d`' need to be specified to the number. If no unit is specified, seconds are assumed.

Examples

EVL job

1. Run EVL job in two steps:

```
Read  file.json INPUT   evd/file.evd --text-input
Map   INPUT     MAPPED  evd/file.evd evd/stage.evd evm/file.evm
Write MAPPED    stage.parquet        evd/stage.evd
```

```
Wait "for creation of parquet file."

Run   "impala \"refresh table stage;\""

End
```

EVL workflow

2. Wait for myself (i.e. the same workflow yesterday) to finish:

   ```
   Wait --myself
   ```

   or the inother words:

   ```
   Wait --myself --yesterday
   ```

   or shortly also:

   ```
   Wait -my
   ```

3. Wait for all jobs to finish, but at most 1 hour, then fail:

   ```
   Run file2stage.bills.evl update.bills.evl
   Run file2stage.invoices.evl
   Wait --time 1h
   ```

4. Wait (forever) for successful run of the job '`file2stage_example.evl`', then continue:

   ```
   Wait job/file2stage.example.evl
   ```

5. Wait (at most 1 day) for the job '`export_job.evl`' before continue:

   ```
   Wait job/export_job.evl --time 1d
   ```

6. Wait (at most 120 minutes) for the job '`sftp_billing.evl`' of the different project '`billing`' (of current ODATE):

   ```
   Wait job/sftp_billing.evl --project billing --time 120m
   ```

7. Wait (at most 300 seconds) for the job '`sftp_billing.evl`' of the different project '`billing`' of the 20260121}:

   ```
   Wait job/sftp_billing.evl --project billing --odate "20260121" --time 300
   ```

8. Wait (at most 6 hours) for previous run of given workflow. (Variable '`ODATE_MINUS1`' has to be set by you.):

   ```
   Wait other_workflow.ewf --odate $ODATE_MINUS1 --time 6h
   ```

Variables Index

General Index

(Index is nonexistent)