laika Documentation

Release 1.2.0

May 24, 2019

Contents

1 Installation **3**

2 Usage **5**
2.1 Arguments 5

3 Testing **7**

4 Documentation Contents **9**
4.1 Configuration 9
4.2 Reports 11
4.3 Results 19
4.4 Reports templating 23

laika is a business reporting library that allows you to request data from different sources and send it to someone or save it at some destination. For example: you can query your database, send the result as an excel attachment via email and save it on Google Drive or Amazon S3.

Check out the documentation at readthedocs.

Laika was tested on Python 2.7 and 3.5 or higher.

CHAPTER 1

Installation

You can install it directly using *pip*:

```
$ pip install laika-lib
```

You can specify extra dependencies. To find out what dependencies you need to install, check out configuration docs. For example, to install libraries to use Google Drive and Amazon S3 in your reports you must run:

```
$ pip install laika-lib[drive, s3]
```

CHAPTER 2

Usage

`laika.py` is a script that lets you use laika library. You can run it like this:

```
$ laika.py some_report
```

This command will run the report named *some_report*. This report must be defined in some configuration file. By default laika looks for `config.json` in the same directory. You can specify a custom config passing `-c` parameter:

```
$ laika.py -c my_config.json
```

Path to configuration file can also be specified with the `LAIKA_CONFIG_FILE_PATH` environment variable:

```
$ export LAIKA_CONFIG_FILE_PATH='/home/me/my_config.json'
$ laika.py my_report
```

Another parameter you can use is `--pwd` which serves for specifying working directory. It can also be specified in configuration file or `LAIKA_PWD` environment variable.

2.1 Arguments

You can check all the predefined `laika.py` arguments with `--help`.

Undefined arguments will be added to report's definition overwriting default values. Thus, if for example the configuration for `my_report` defines field `my_field` with value `foo`, if you execute it like this:

```
$ laika.py my_report --my_field bar
```

`my_field` configuration will contain `bar` as value.

You can also overwrite *Global configuration* via command line arguments.

CHAPTER 3

Testing

To run test, you must install test dependencies:

```
$ pip install laika-lib[test]
```

Then, run test from laika directory:

```
$ cd path/to/laika
$ python -m unittest discover
```

CHAPTER 4

Documentation Contents

4.1 Configuration

Laika reads reports definitions from a json file which must have this structure:

```
{
  "include": [...],
  "profiles": [...],
  "connections": [...],
  "reports": [...]
}
```

The configuration can be separated in multiple files. In this case there must be a base configuration file that will have to include the other files via `"include"` field with a list of paths:

```
"include": [
  "another_config.json",
  "/some/config.json"
]
```

These files will be included in the configuration. The only constraint is they can only have `reports`, `connections` and `profiles` fields defined.

You can check the example configuration file for more information.

4.1.1 Profiles

Profiles are all kind of credentials used for accessing external APIs (like Google Drive). You must specify a name and a path to credentials for each profile. For example:

```
{
  "name": "my_drive",
```

(continues on next page)

(continued from previous page)

```
  "credentials": "secret.json"
}
```

`credentials` is always a path to a json file, but it's format depends on each type of report or result. For example email credentials are defined like this:

```
{
  "username": "me@gmail.com",
  "password": "secret"
}
```

4.1.2 Connections

Connections are used to access data sources or destinations. They must have a *name* and a *type*, and a set of specific fields. Currently supported connections are described below.

Database

Database connection examples:

Postgres

```
{
  "name": "local",
  "type": "sqlalchemy",
  "constring": "postgresql://user@localhost:5432/database"
}
```

Presto

```
{
  "name": "local",
  "type": "sqlalchemy",
  "constring": "presto://user@localhost:8889/default"
}
```

Email

Example of a smtp connection:

```
{
  "name": "gmail_smtp",
  "type": "email",
  "host": "smtp.gmail.com",
  "port": 587
}
```

Ftp

Example of a ftp connection:

```
{
  "name": "some_ftp",
  "type": "ftp",
  "host": "ftp.home.com"
}
```

4.1.3 Global configuration

In addition to reports, connections and profiles you can define this configurations:

- now: string with a datetime to use as current datetime. Useful if your reports or results make use of templating to depend on dates relative to current date. Must match `%Y-%m-%d %H:%M:%S` format.
- timezone: string of timezone to use. By default all the dates will be generated in UTC. You can overwrite it for each particular report.
- pwd: directory, to which laika will change before executing reports. In this directory it will, for example, read query files, or save file results (if relative path is specified).

These configurations can be overwritten via command line arguments:

```
$ laika.py my_report --now "2018-11-12 00:00:00"
```

4.2 Reports

Reports are defined as json objects that must define these fields:

- name: this name will be used to execute the report through cli.
- type: report's type. Supported report types are defined below.
- results: list of results configuration that define how to save the reports (*Results documentation*).
- Set of required or optional fields that are detailed below.

4.2.1 File

`type: file`. This report reads the data from local file. These are the configurations:

- filename: path to the file. Laika parses the file based on it's extension. *csv* and *tsv* files are parsed out of the box. To parse excel files, you need to install `laika-lib[excel]` dependency.
- raw: if this parameter is set to `true`, file's extension will be ignored and file contents will be passed to result unparsed.

Example of a file report:

```
{
  "name": "my_file_report",
  "type": "file",
  "filename": "/path/to/filename.xlsx",
  "results": [...]
}
```

4.2.2 Query

Note: To use query report you must install `sql` dependency (for Sqlalchemy): `pip install laika-lib[sql]`. You also have the libraries needed to access your specific database.

For postgres: `pip install laika-lib[postgres]`

For Presto(Pyhive): `pip install laika-lib[presto]`

We only tested it with postgres and Presto, but it should work with all databases supported by SQLAlchemy.

`type: query`. This report runs a query to some database. Should work with any database supported by Sqlalchemy but right now it's only tested with PostgreSQL and Presto. These are the configurations:

- query_file: path to a file that contains plane sql code.
- connection: name of the connection to use.
- variables: A dictionary with values to replace in query code. You can find further explanation in *Query templating*.

Example of a query report:

```
{
  "name": "my_shiny_query",
  "type": "query",
  "query_file": "/my/dir/query.sql",
  "connection": "local",
  "variables": {
    "foo": "bar"
  },
  "results": [...]
}
```

4.2.3 Bash Script

`type: bash`. This report executes a bash command and reads it's output. You can interpret this report as the `read_output` part of this example:

```
$ bash_script | read_output
```

These are the configurations:

- script: command to execute
- script_file: path to the file with bash script to execute. If *script* is defined, this field will be ignored.
- result_type: type of output data format. Can be *csv*, *json* or *raw*. In case of *raw*, the content will not be converted and will be passed as is to the result. The explanation of *json* format is explained below.

Example bash script report:

```
{
  "name": "some_bash_script",
  "type": "bash",
  "script_file": "some_script.sh",
  "result_type": "json",
```

(continues on next page)

(continued from previous page)

```
  "results": [...]
}
```

Bash Script json format

Json data will be converted to a pandas dataframe using `pd.read_json` function (Docs). These are some examples of the formats it accept:

Example 1 (all arrays must have the same size):

```
{
  "column_1": ["data_row_1", "data_row_2", "data_row_3"],
  "column_2": ["data_row_1", "data_row_2", "data_row_3"],
  ...
}
```

Example 2:

```
[
  {
    "column_1": "data_row_1",
    "column_2": "data_row_1",
    "column_3": "data_row_1",
  },
  {
    "column_1": "data_row_2",
    "column_3": "data_row_2"
  }
  ...
]
```

4.2.4 Download From Google Drive

Note: To use drive report you must install `drive` dependency: `pip install laika-lib[drive]`

`type: drive`. This report downloads a file from Google Drive. It uses file parsing logic from the File report.

Configuration:

- profile: Name of the profile to use. It's credentials must be ones of a service account with access to Google Drive's API.
- grant: email of a grant account, in the name of which the document will be downloaded. Grant must have access to specified folder.
- filename: name of the file to download.
- folder: directory in which the report will search for the specified file.
- folder_id: google drive's id of the above folder. If specified, folder option is ignored. It's useful if there is more than one folder with the same name.
- subfolder: optional, if specified, this report will look for a subfolder inside a folder and, if found, will look there for filename. In other words, it will look for this structure: `<folder>/<subfolder>/<filename>`

- file_id: google drive's id of the file to download. If specified, all other file options are ignored.

Example of a drive report:

```
{
  "type": "drive",
  "profile": "drive_service_account",
  "grant": "me@mail.com",
  "folder_id": "my_folder_id",
  "folder": "TestFolder",
  "subfolder": "TestSubFolder",
  "file_id": "my_file_id",
  "filename":"file_to_download.xlsx"
}
```

4.2.5 Download From S3

Note: To use S3 report you must install `s3` dependency: `pip install laika-lib[s3]`

`type: s3`. This report downloads a file from Amazon S3. It uses file parsing logic from the File report. In order to use this report, you have to install boto3.

Configuration:

- profile: Name of profile to use (laika profile, no to confuse with aws profiles). Credentials file of the specified profile must contain data to be passed to Session constructor. Example of a minimal aws credentials file for laika:

`json { "aws_access_key_id": "my key id", "aws_secret_access_key": "my secret access key" }`

- bucket: s3 bucket to download the file from.
- filename: File to download. This config is the *key* of the file in bucket.

Example of a s3 report:

```
{
  "name": "s3_report_example",
  "type": "s3",
  "profile": "my_aws_profile",
  "bucket": "some.bucket",
  "filename": "reports/custom_report.csv",
  "results": [...]
}
```

4.2.6 Redash

`type: redash`. This report downloads query result from redash API. These are the configurations:

- redash_url: the url of your redash instance.
- query_id: id of the query to download. You can get from the query's url, it's last part is the id (for example, for `https://some.redash.com/queries/67`, 67 is the id).
- api_key: token to access the query, either for user or for query. You can find user's token in the profile, token for query can be found in the source page.

- refresh: True if you want an updated report. **Important**: For refresh to work the api_key must be of user type.
- parameters: Dictionary of query parameters. They should be written as they are defined in the query. The p_ needed for the url will be prepended in the report.

Example of a redash query:

```
{
  "name": "some_redash_query",
  "type": "redash",
  "api_key": "some api key",
  "query_id": "123",
  "redash_url": "https://some.redash.com",
  "refresh": true,
  "parameters": {
      "hello": "world"
  },
  "results": [...]
}
```

4.2.7 Adwords

Note: To use adwords report you must install `adwords` dependency: `pip install laika-lib[adwords]`

`type: adwords`. This report is generated by Google Adwords API. To use it, you will need to install googleads. The configurations are:

- profile: Name of profile to use. Credentials file is a *.yaml*, you can find out how to generate it in adwords API tutorial.
- report_definition: the definition of the report which will be passed to DownloadReport method of googleads API. You will normally define fields `reportName`, `dateRangeType`, `reportType`, `downloadFormat`, `selector`, but these will vary depending on the report type.
- reportName: In order not to repeat reports definitions, you can specify this name and reuse the definition. In other words, you can have multiple reports with the same name, but only one report_definition, which will be used for all of them.
- dateRangeType: if you use report_definition from another report, you can overwrite date range it uses with this configuration. Here you can read more about date range types you can chose from.
- date_range: if dateRangeType is set to `CUSTOM_DATE`, you can define a custom range of dates to extract. The definition must be a dictionary with min and max values. In both you can use relative dates with *Filenames templating*.
- client_customer_id. Id or list of ids of adwords customers, whose data you want in the report.

Example of adwords query:

```
{
  "name": "some_adwords_report",
  "type": "adwords",
  "date_range": {"min": "{Y-1d}{m-1d}{d-1d}", "max": "{Y-1d}{m-1d}{d-1d}"},
  "client_customer_ids": "123-456-7890",
  "report_definition": {
    "reportName": "Shopping Performance Last Month",
    "dateRangeType": "CUSTOM_DATE",
```

(continues on next page)

(continued from previous page)

```
    "reportType": "SHOPPING_PERFORMANCE_REPORT",
    "downloadFormat": "CSV",
    "selector": {
        "fields": [
            "AccountDescriptiveName",
            "CampaignId",
            "CampaignName",
            "AdGroupName",
            "AdGroupId",
            "Clicks",
            "Impressions",
            "AverageCpc",
            "Cost",
            "ConvertedClicks",
            "CrossDeviceConversions",
            "SearchImpressionShare",
            "SearchClickShare",
            "CustomAttribute1",
            "CustomAttribute2",
            "Brand"
        ]
    }
  },
  "results": [...]
}
```

4.2.8 Facebook Insights

`type: facebook`. Retrieves the data from the Facebook's Insights API. The report is requested as asynchronous job and is polled for completion every few seconds.

Configuration:

- profile: Name of profile to use. Credentials file must contain access token with at least `read_insights` permission. You can generate it in Facebook's developers panel for you app. Example `facebook` credentials:

```
{
  "access_token": "..."
}
```

- object_id: Facebook's object id from which you want to obtain the data.
- params: Set of parameters that will be added to the request. Check the example report to know what values are used by default, consult Facebook's Insights API documentation to discover what parameters you can use.
- sleep_per_tick: Number of seconds to wait between requests to Facebook API to check if the job is finished.
- since: Starting date for a custom date range. Will only be used if `date_preset`, `time_range` or `time_ranges` are not present among report parameters. You can set relative dates using *Filenames templating*.
- until: Same as since, but for the ending date.

Example of facebook report:

```
{
    "name": "my_facebook_insights_report",
```

(continues on next page)

(continued from previous page)

```
    "type": "facebook",
    "profile": "my_facebook_profile",
    "object_id": "foo_1234567890123456",
    "since": "{Y-1d}-{m-1d}-{d-1d}",
    "until": "{Y-1d}-{m-1d}-{d-1d}",
    "params": {
        'level': 'ad',
        'limit': 10000000,
        'filtering': '[{"operator": "NOT_IN", "field": "ad.effective_status", "value
↪": ["DELETED"]}]',
        'fields': 'impressions,reach',
        'action_attribution_windows': '28d_click'
    },
    "results": [...]
}
```

4.2.9 RTBHouse

Note: To use rtbhouse report you must install `rtbhouse` dependency: `pip install laika-lib[rtbhouse]`

`type: rtbhouse`. Downloads marketing costs report from RTBHouse API. Reported campaigns (advertisers) are all those created by the account.

Configuration:

- profile: Name of profile to use. Credentials file must be a json containing `username` and `password` fields.
- day_from: Starting date for the period to retrieve costs for. You can set a relative date using *Filenames templating*.
- day_to: Same as day_from, but for the ending date.
- group_by and convention_type: Optional parameters to send to RTBHouse.
- campaign_names: Mapping from campaign hash to a readable name for the resulting report.
- column_names: Mapping to rename columns in the resulting report.

Example of rtbhouse report:

```
{
  "name": "my_rtbhouse_report",
  "type": "rtbhouse",
  "profile": "my_rtbhouse_profile",
  "group_by": "day",
  "convention_type": "ATTRIBUTED",
  "day_from": "{Y-1d}-{m-1d}-{d-1d}",
  "day_to": "{Y-1d}-{m-1d}-{d-1d}",
  "campaign_names": {
    "1234567890": "Some readable campaign name"
  },
  "column_names": {
    "hash": "CampaignID",
    "name": "Campaign",
    "campaignCost": "Cost",
```

(continues on next page)

(continued from previous page)

```
    "day": "Date"
  },
  "results": [...]
}
```

4.2.10 Rakuten

`type: rakuten`. Downloads a report from Rakuten marketing platform by name.

Configuration:

- profile: Name of profile to use. Credentials file must be a json containing `token` key, with a token to access Rakuten API.
- report_name: Existing report to download from the platform.
- filters: A set of filters to send to the API. Must be a dictionay, you can use *Filenames templating* on the values.

Example of rakuten report:

```
{
  "name": "my_rakuten_report",
  "type": "rakuten",
  "profile": "my_rakuten_profile",
  "report_name": "some-report",
  "filters": {
    "start_date": "{Y-10d}-{m-10d}-{d-10d}",
    "end_date": "{Y-1d}-{m-1d}-{d-1d}",
    "include_summary": "N",
    "date_type": "transaction"
  }
}
```

4.2.11 Module

`type: module`. Allows you to use a python module with custom report class to obtain the data. This module will be loaded dynamically and executed. Currently it has the same configuration as the module result, which can be confusing.

Configuration:

- result_file: Path to python file.
- result_class: Name of the class to use as result inside the python file. This class must inherit `Report` class and define `process` method, which should normally return report data. Simple example of a custom report class:

```
from laika.reports import Report

class FooResult(Report):

    def process(self):
        # using some custom configs
        filename = self.custom_filename
        # returning some data
        with open(filename) as f:
            return do_stuff(f.read())
```

This report will be executed as any other report - it will have available all the extra configuration you define.

> **Warning:** This report will load and execute arbitrary code, which implies a series of security holes. Always check custom modules before using them.

Example of a module report definition:

```
{
  "type": "module",
  "result_file": "./some_folder/my_custom_report.py",
  "result_class": "MyReport",
  "my_custom_config": "value"
}
```

4.3 Results

Results are defined for each report in a list. Each result is an object, that must define *type* field. The rest of the fields depend on the type of result. Below are described all the supported results.

> **Note:** Results will be executed in the same order they are defined. If the first one raises an exception, the execution will be suspended (the next ones will not execute).

4.3.1 File

`type: file`. This result will save the data as a file. The configurations for this result are:

- filename: path to the file. Depending on the file extension this file will be saved as excel (xls or xlsx), tsv or csv.

Example of a file result:

```
"result": {
  "type": "file",
  "filename": "output.csv"
}
```

4.3.2 Email

`type: email`. Sends the data as an attachment in an email. Configurations for this result are:

- connection: Name of a connection with type *email*.
- profile: Profile name to use. Credentials from this profile must have `username` and `password` field from smtp service you will use.
- filename: name of the file to attach.
- recipients: one or more recipients for the email. Can be a string with one recipient or a list of recipients. This camp is required.
- subject: subject of the email.
- body: text the email will contain.

- attachments: Optional. list of files to attach to the email. Must be a list of paths to files.

Subject and body can be formatted the same way filenames are formatted. You can find more about it in *Filenames templating*. Example of email result:

```
{
  "type": "email",
  "filename": "report_{Y}-{m}-{d}_{H}-{M}.xlsx",
  "recipients": ["some_recipient@mail.com", "another_recipient@foo.com"],
  "subject": "Some report",
  "body": "Report generated {Y}-{m}-{d} {H}:{M}. \nDo not reply this email!",
  "profile": "my_gmail",
  "connection": "gmail_smtp"
}
```

4.3.3 Ftp

`type: ftp`. Uploads the data to a ftp server. Configurations for this result are:

- profile: Name of the profile to use. Credentials must have `username` and `password` fields to authenticate in the ftp service.
- connection: Name of a connection of ftp type.
- filename: Name with which the file will be uploaded to ftp.

Example of ftp result:

```
{
  "type": "ftp",
  "profile": "my_ftp_profile",
  "connection": "some_ftp",
  "filename": "my_report.csv"
}
```

4.3.4 Google Drive

Note: To use drive result you must install `drive` dependency: `pip install laika-lib[drive]`

`type: drive`. Saves report data in Google Drive. These are the configurations:

- profile: Name of the profile to use. Credentials of this profile must be ones of a service account with access to Google Drive API.
- filename: Name for the resulting file.
- folder: Directory in which the file will be stored. If not specified, the file is stored in the root of given drive. If there is more than one directory with the same name, the file will be stored in the first one this result finds (depends on Drive API).
- folder_id: Id of the directory in which the result will be saved. If specified, *folder* configuration will be ignored. You can get this id from the url in Google Drive web interface.
- grant: Email of user, in the name of whom the file will be uploaded. Must have access to specified folder.
- mime_type: Media type of the file to be uploaded. If none is specified it will take the type of the filename extension.

Example of drive result:

```
{
  "type": "drive",
  "profile": "my_service_drive_account",
  "filename": "report.xlsx",
  "folder": "TestFolder",
  "grant": "me@mail.com"
}
```

4.3.5 Amazon S3

Note: To use S3 result you must install `drive` dependency: `pip install laika-lib[drive]`

`type: s3`. Saves the result in Amazon S3. In order to use this result, you have to install boto3.

Configuration:

- profile: Name of profile to use (laika profile, no to confuse with aws profiles). Credentials file of the specified profile must contain data to be passed to Session constructor. Example of a minimal aws credentials file for laika:

`json { "aws_access_key_id": "my key id", "aws_secret_access_key": "my secret access key" }`

- bucket: s3 bucket in which you want to save your data.
- filename: Name of the file to save. This config is the *key* of the file in bucket.

Example of s3 result:

```
{
  "type": "s3",
  "profile": "my_aws_profile",
  "bucket": "some.bucket",
  "filename": "reports/custom_report.csv"
}
```

4.3.6 SFTP

`type: sftp`. Uploads the data to a SFTP server. Configurations for this kind of result are:

- profile: Name of the profile to use. Credentials file must have `username` and optionally `password` fields and/or `private_key` to authenticate in the SFTP service. `private_key` should be a path to a file with the private key.
- connection: Name of a connection of ftp type.
- folder: Folder in which the file will be saved. Can be a unix style path.
- filename: Name with which the file will be uploaded to ftp.

Example of SFTP result:

```
{
  "type": "sftp",
  "profile": "my_sftp_profile",
  "connection": "some_sftp",
  "folder": "./some_folder/",
  "filename": "my_report.csv"
}
```

4.3.7 Redash

`type: redash`. Saves the data as *json* file in format which redash understands. You can then expose it to redash via API, redash will be able to consume it using url datasource. Configuration has the same fields as *File* result, with the exception of the fact that the file must be json (it will be saved as json, regardless of the extension).

4.3.8 Fixed Columnar Result

`type: fixed`. Wrapper result that ensures the presence of a list of columns in the data before sending them to an inner result. Columns not present in the data will be added. Can only be used with reports that return a `pandas.DataFrame` as result (or some data structure accepted by DataFrame's constructor). All the configuration keys, besides ones this result defines, will be passed to the inner result. Can be useful if you need to adapt the data to some external format (i.e. Hive schema).

Configuration:

- columns: List of columns to leave in the data, in the order you want them to appear for the inner result.
- inner_result_type: Type of result to use after fixing the data.
- default_value: This value will be used to fill missing columns with (`np.nan` by default).

Example of fixed columnar result:

```
{
  "type": "fixed",
  "columns": ["id", "date", "action", "value", "missing_column"],
  "default_value": "value_to_fill_missing_column_with",
  "inner_result_type": "file",
  "filename": "resulting_output.csv"
}
```

As you can see in the example, you define both configurations for the fixed columnar result, and the result it wraps (in this case a file result, with it's corresponding filename). Only the columns defined in the configuration will be passed to the inner result.

4.3.9 Module

`type: module`. Allows you to use a python module with custom result class to save the data. This module will be loaded dynamically and executed.

Configuration:

- result_file: Path to python file.
- result_class: Name of the class to use as result inside the python file. This class must inherit `Result` class and define `save` method. Simple example of a custom result class:

```
from laika.reports import Result

class FooResult(Result):

    def save(self):
        # using some custom configs
        filename = self.custom_filename
        # doing the actual save
        print str(self.data)
```

This result will be executed as any other result - it will have available all the extra configuration you define.

Warning: this result will load and execute arbitrary code, which implies a series of security holes. Always check custom modules before using them.

Example of a module result definition:

```
{
  "type": "module",
  "result_file": "./some_folder/my_custom_result.py",
  "result_class": "MyResult",
  "my_custom_config": "value"
}
```

4.4 Reports templating

In query definitions (or other templates inside laika) you can specify dynamic dates this way:

```
select * from some_table where date >= '{m}' and date < '{m+1m}'
```

laika will replace this dates by (supposing current month is February of 2016):

```
select * from some_table where date >= '2016-02-01 00:00:00' and date < '2016-03-01␣
↪00:00:00'
```

Dates are UTC by default, but you can modify that changing `timezone` configuration.

These are all the template variables you can use:

- `{now}`: current date.
- `{d}` o `{t}`: start of current date (00:00:00 of today)
- `{m}`: start of first day of current month
- `{y}`: start of current year (first day of January)
- `{H}`: start of current hour
- `{M}`: start of current minute
- `{w}`: start of current week (Monday)

These variables may also receive modifiers. Modifier expression must start with one of these variables, continue with a sign (+ o −), a number and finally, a measure. This measures can be:

- `{y}`: years

- {m}: months
- {d}: days
- {h}: hours
- {M}: minutes
- {w}: weeks

For example:

```
{now}, {now-1d}, {now+1y}, {now+15h}, {t-3m}
```

Results in:

```
2016-02-12 18:19:09, 2016-02-11 18:19:09, 2017-02-12 18:19:09, 2016-02-13 09:19:09,␣
↪2015-11-12 00:00:00
```

Another possibility is to specify a start of week with {f}. For example, {d-1f} will move the date to Monday of the current week, and {d+2f} will move the date to Monday within two weeks.

4.4.1 Query templating

If the report has a dictionary of variables specified they will be replaced in the specified query file. For example, if you define a query like this:

```
select something from some_table where type = '{my_type}'
```

You can then pass the variables through configuration this way:

```
{
  "variables": {
    "my_type": "some custom type"
  }
}
```

The query that will end up executing is this:

```
select something from some_table where type = 'some custom type'
```

These variables will be replaced first, and then laika will replace the dates, so you can define in your configuration variables like this:

```
{
  "variables": {
    "yesterday": "{t-1d}"
  }
}
```

{yesterday} will be converted into 2016-02-12 17:19:09.

4.4.2 Filenames templating

filename configuration in all the reports and results can be formatted in a similar way. For example, if you specify:

```
{
  "filename": "report_{Y}-{m}"
}
```

This will be formatted as `report_2016-02` (assuming the report ran in February of 2016).

You can also use the same modifiers:

```
{
  "filename": "report_{Y}-{m-1m}"
}
```

Will result in `report_2016-01`.