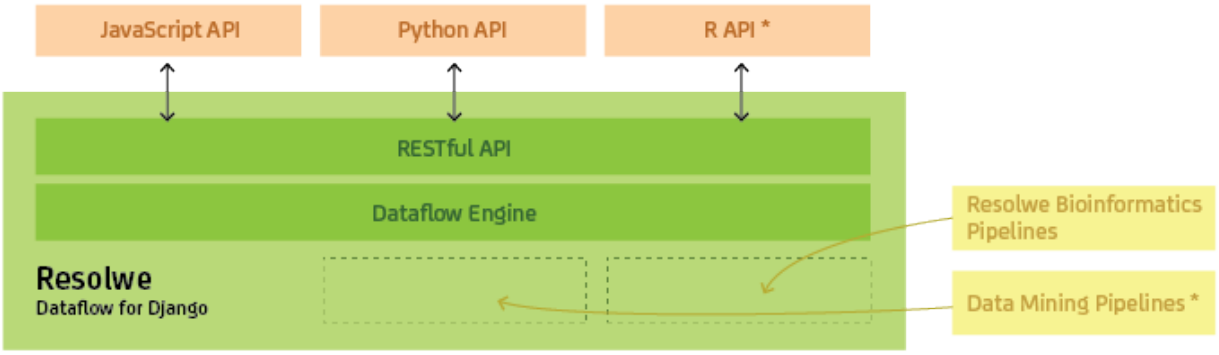
Resolwe

Release 20.0.0a2.dev15+gd53ab55

Jan 22, 2020

Contents

Resolwe is an open source dataflow package for Django framework. It offers a complete RESTful API to connect with external resources. A higher layer of convenience APIs for JavaScript, Python and R are in development. A collection of bioinformatics pipelines is available within the Resolwe Bioinformatics project. We envision a similar toolkit for machine learning.

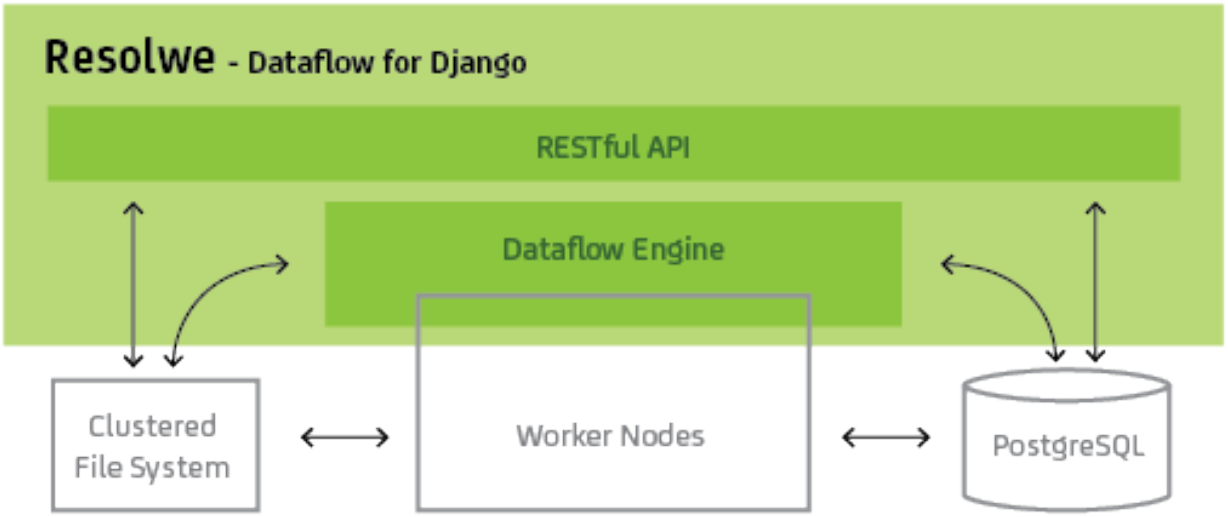


CHAPTER 1

Contents

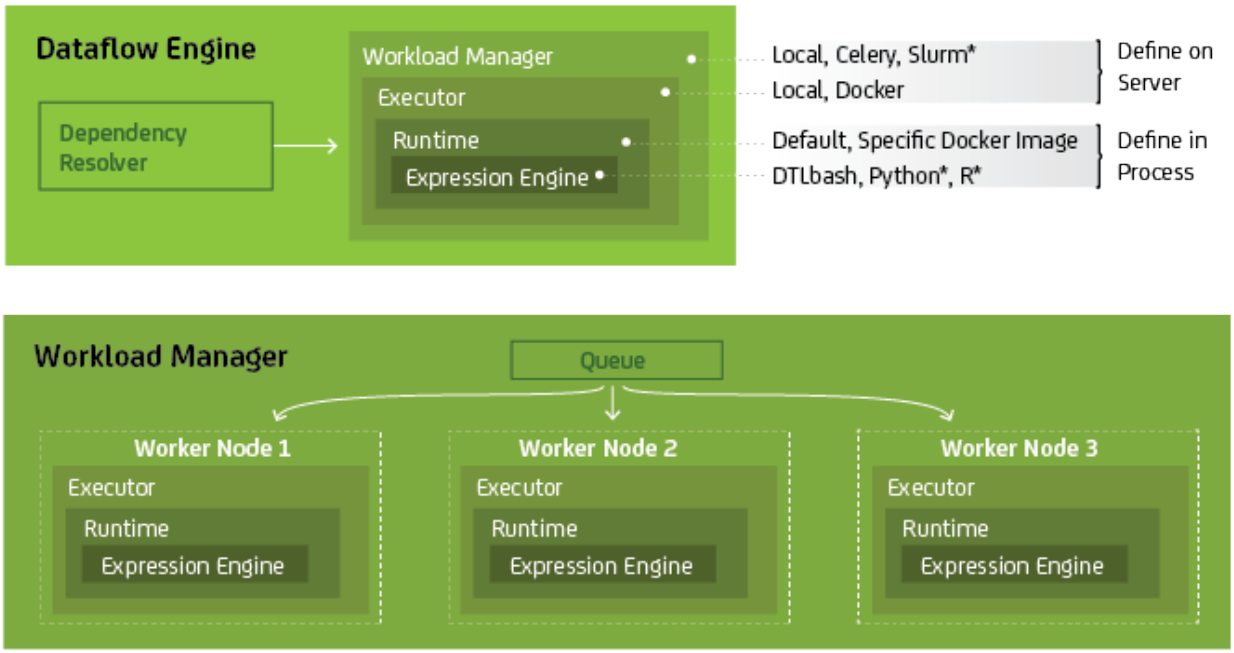
1.1 Overview

Resolwe consists of two major components: a RESTful API and the Flow Engine. The RESTful API is based on the Django REST Framework and offers complete control over the workflow, the data involved and the permissions on those data. The Resolwe Flow engine, on the other hand, handles pipeline execution. It resolves dependencies between *processes* (jobs or tasks), and executes them on worker nodes. Results are saved to a PostgreSQL database and a clustered file system.



The Flow Engine has several layers of execution that can be configured either on the server or by the individual processes.

Processes can be executed on a server cluster. In this case the Executor, Runtime and Expression Engine layers span multiple worker nodes.



Resolwe can be configured for lightweight desktop use (*e.g.,* by bioinformatics professionals) or deployed as a complex set-up of multiple servers and worker nodes. In addition to the components described above, customizing the configuration of the web server (*e.g.,* NGINX or Apache HTTP), workload manager, and the database offer high scaling potential.

Example of a lightweight configuration: synchronous workload manager that runs locally, Docker executor and runtime, Django web server, and local file system.

Example of a complex deploy: Slurm workload manager with a range of computational nodes, Docker executor and runtime on each worker node, NGINX web server, and a fast file system shared between worker nodes.

1.2 Getting started

TODO: Write about how to include Resolwe in a Django project and explain settings parameters. Create an example Django project in the `docs/example` folder and give code references where a detailed explanation is needed.

1.3 Writing processes

Process is a central building block of the Resolwe's dataflow. Formally, a process is an algorithm that transforms inputs to outputs. For example, a *Word Count* process would take a text file as input and report the number of words on the output.

When you execute the process, Resolwe creates a new `Data` object with information about the process instance. In this case the *document* and the *words* would be saved to the same `Data` object. What if you would like to execute another analysis on the same document, say count the number of lines? We could create a similar process *Number of Lines* that would also take the file and report the number of lines. However, when we would execute the process we would have 2 copies of the same *document* file stored on the platform. In most cases it makes sense to split the upload (data storage) from the analysis. For example, we could create 3 processes: *Upload Document*, *Word Count* and *Number of Lines*.

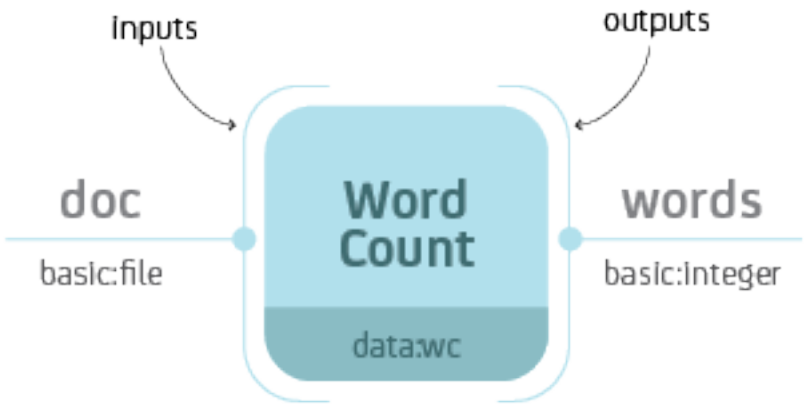


Fig. 1.1: *Word Count* process with input `doc` of type `basic:file` and output `words` of type `basic:integer`.

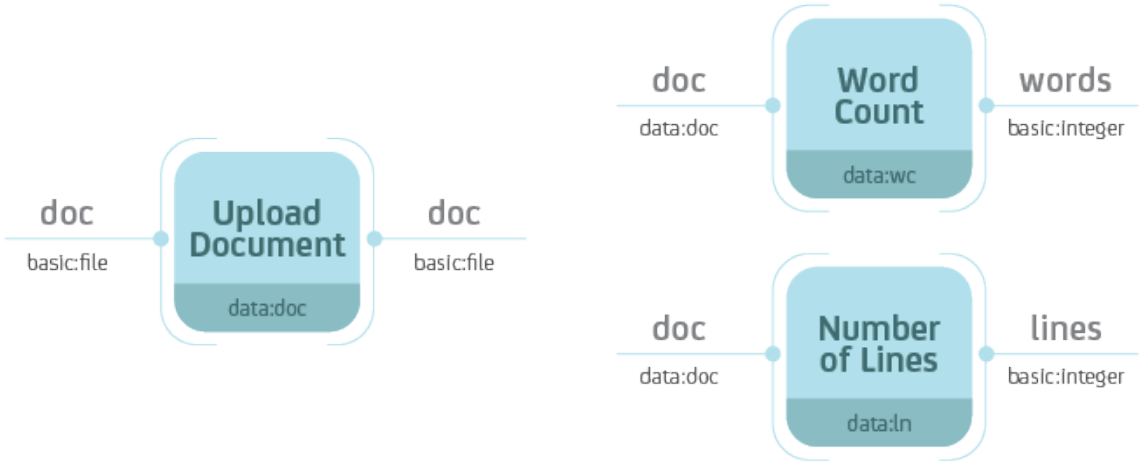


Fig. 1.2: Separate the data storage (*Upload Document*) and analysis (*Word Count*, *Number of Lines*). Notice that the *Word Count* and *Number of Lines* processes accept `Data` objects of type `data:doc`—the type ot the *Upload Document* process.

Resolwe handles the execution of the dataflow automatically. If you were to execute all three processes at the same time, Resolwe would delay the execution of *Word Count* and *Number of Lines* until the completion of *Upload Document*. Resolwe resolves dependencies between processes.

A processes is defined by:

- Inputs
- Outputs
- Meta-data
- Algorithm

Processes are stored in the data base in the `Process` model. A process' algorithm runs automatically when you create a new `Data` object. The inputs and the process name are required at `Data` create, the outputs are saved by the algorithm, and users can update the meta-data at any time. The *Process syntax* chapter explains how to add a process definition to the `Process` data base model

Processes can be chained into a dataflow. Each process is assigned a type (*e.g.,* `data:wc`). The `Data` object created by a process is implicitly assigned a type of that process. When you define a new process, you can specify which data types are required on the input. In the figure below, the *Word Count* process accepts `Data` objects of type `data:doc` on the input. Types are hierarchical with each level of the hierarchy separated by a colon. For instance, `data:doc:text` would be a sub-type of `data:doc`. A process that accepts `Data` objects of type `data:doc`, also accepts `Data` objects of type `data:doc:text`. However, a process that accepts `Data` objects of type `data:doc:text`, does not accept `Data` objects of type `data:doc`.

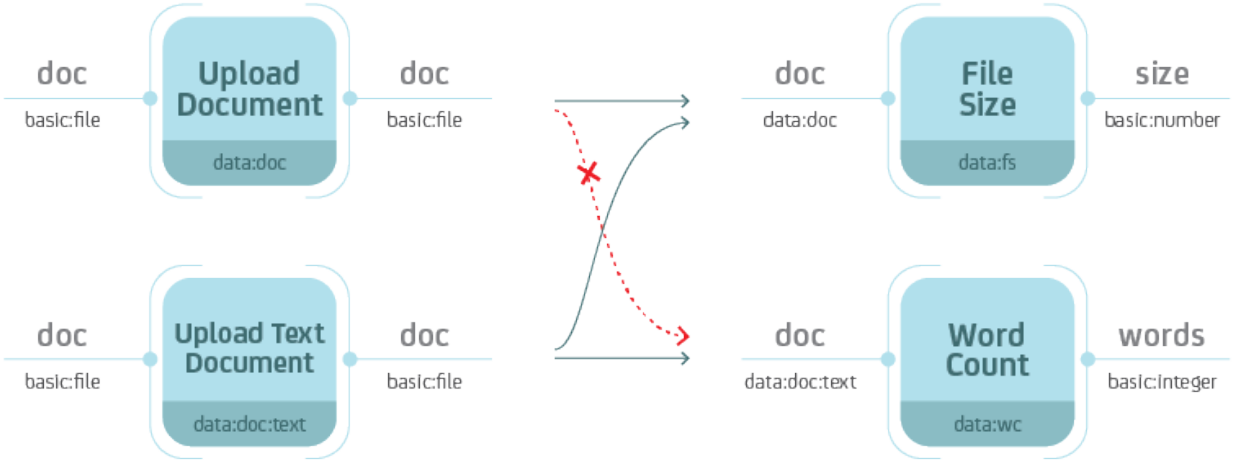


Fig. 1.3: Types are hierarchical. When you define the type on the input, keep in mind that the process should also handle all sub-types.

1.3.1 Process syntax

A process can be written in any syntax as long as you can save it to the `Process` model. The most straight-forward would be to write in Python, using the Django ORM:

```
p = Process(name='Word Cound',
            slug='wc-basic',
            type='data:wc:',
            inputs = [{
                'name': 'document',
                'type': 'basic:file:'
```

```
            }],
            outputs = [{
                'name': 'words',
                'type': 'basic:integer:'
            }],
            run = {
                'bash': 'WORDS=`wc {{ document.file }}\n`' +
                        'echo {"words": $WORDS}'
            })
p.save()
```

We suggest to write processes in the YAML syntax. Resolwe includes a `register` Django command that parses .yml files in the `processes` directory and adds the discovered processes to the `Process` model:

```
./manage.py register
```

Do not forget to re-register the process after you make changes to the .yml file. You have to increase the process version each time you register it. For development, you can use the `--force` option (or `-f` for short):

```
./manage.py register -f
```

This is an example of `the smallest processor` in YAML syntax:

```
- slug: mini
  name: Minimalistic Process
  requirements:
    expression-engine: jinja
  type: "data:mini"
  run:
    language: bash
    program: |
      echo 'Hello bioinformatician!'
```

This is the example of the `basic Word Count` implementation in the YAML syntax (with the document file as input):

```
- name: Word Count
  slug: wc-basic
  type: "data:wc"
  inputs:
    - name: document
      type: basic:file
  outputs:
    - name: words
      type: basic:integer
  run:
    language: bash
    program: |
      WORDS=$(wc {{ document.file }})
      echo {"words": $WORDS}
```

If you would like to review the examples of the three processes mentioned above (*Upload Document*, *Word Count* and *Number of Lines*), `follow this link`. Read more about the process options in *Process schema* below.

1.3.2 Process schema

Process is defined by a set of fields in the `Process` model. We will describe how to write the process schema in YAML syntax. Some fields in the YAML syntax have different name or values than the actual fields in the `Process` model. See an `example of a process with all fields`. Fields in a process schema:

Field	Short description	Required	Default
slug	unique id	required	
name	human readable name	required	
description	detailed description	optional	<empty string>
version	version numbering	optional	
type	data type	required	
category	menu category	optional	<empty string>
entity	automatic grouping	optional	
persistence	storage optimization	optional	RAW
scheduling_class	scheduling class	optional	batch
input	list of input fields	optional	<empty list>
output	list of result fields	optional	<empty list>
run	the algorithm	required	
requirements	requirements	optional	<empty dict>

Slug

TODO

Name

TODO

Description

TODO

Version

TODO

Type

TODO

Category

The category is used to arrange processes in a GUI. A category can be any string of lowercase letters, numbers, - and :. The colon is used to split categories into sub-categories (*e.g.,* `analyses:alignment`).

We have predefined three top categories: upload, import and analyses. Processes without this top category will not be displayed in the GenBoard interface, but will be available on the platform.

Entity

With defining the `entity` field in the process, new data objects will be automatically attached to a new or existing Entity, depending on it's parents and the definition of the field.

`entity` field has 3 subfields:

- `type` is required and defines the type of entity that the new `Data` object is attached to
- `input` limits the group of parents' enteties to a single field (dot separated path to the field in the definition of input)
- `descriptor_schema` specifies the slug of the descriptor schema that is attached to newly created entity. It defaults to the value of `type`

Persistence

Use RAW for imports. CACHED or TMP processes should be idempotent.

Scheduling class

The scheduling class specifies how the process should be treated by the scheduler. There are two possible values:

- `batch` is for long running tasks, which require high throughput.
- `interactive` is for short running tasks, which require low latency. Processes in this scheduling class are given a limited amount of time to execute (default: 30 seconds).

The default value for processes is `batch`.

Input and Output

A list of *Resolwe Fields* that define the inputs and outputs of a process. A *Resolwe Field* is defined as a dictionary of the following properties:

Required *Resolwe Field* properties:

- `name` - unique name of the field
- `label` - human readable name
- `type` - type of field (either `basic:<...>` or `data:<...>`)

Optional *Resolwe Field* properties (except for `group`):

- `description` - displayed under titles or as a tooltip
- `required` - (choices: *true*, *false*)
- `disabled` - (choices: *true*, *false*)
- `hidden` - (choices: *true*, *false*)
- `default` - initial value
- `placeholder` - placeholder value displayed if nothing is specified
- `validate_regex` - client-side validation with regular expression
- `choices` - list of choices to select from (`label`, `value` pairs)

Optional *Resolwe Field* properties for `group` fields:

- `description` - displayed under titles or as a tooltip
- `disabled` - (choices: *true*, *false*)
- `hidden` - (choices: *true*, *false*)
- `collapsed` - (choices: *true*, *false*)
- `group` - list of process fields

TODO: explain what is field schema. For field schema details see fieldSchema.json.

Run

The algorithm that transforms inputs into outputs. Bash and workflow languages are currently supported and we envision more language support in the future (*e.g.,* directly writing processes in Python or R). Commands should be written to a `program` subfield.

TODO: link a few lines from the all_fields.yml process

Requirements

A dictionary defining optional features that should be available in order for the process to run. There are several different types of requirements that may be specified:

- `expression-engine` defines the name of the engine that should be used to evaluate expressions embedded in the `run` section. Currently, only the `jinja` expression engine is supported. By default no expression engine is set, so expressions cannot be used and will be ignored.
- `executor` defines executor-specific options. The value should be a dictionary, where each key defines requirements for a specific executor. The following executor requirements are available:
  - `docker`:
    - `image` defines the name of the Docker container image that the process should run under.
- `resources` define resources that should be made available to the process. The following resources may be requested:
  - `cores` defines the number of CPU cores available to the process. By default, this value is set to `1` core.
  - `memory` defines the amount of memory (in megabytes) that the process may use. By default, this value is set to `4096` MiB.
  - `network` should be a boolean value, specifying whether the process requires network access. By default this value is `false`.

1.3.3 Types

Types are defined for processes and *Resolwe Fields*. `Data` objects have implicitly defined types, based on the corresponding processor. Types define the type of objects that are passed as inputs to the process or saved as outputs of the process. Resolwe uses 2 kinds of types:

- `basic:`
- `data:`

`Basic:` types are defined by Resolwe and represent the data building blocks. `Data:` types are defined by processes. In terms of programming languages you could think of `basic:` as primitive types (like integer, float or boolean) and of `data:` types as classes.

Resolwe matches inputs based on the type. Types are hierarchical, so the same or more specific inputs are matched. For example:

- `data:genome:fasta:` will match the `data:genome:` input, but
- `data:genome:` will not match the `data:genome:fasta:` input.

Note: Types in a process schema do not have to end with a colon. The last colon can be omitted for readability and is added automatically by Resolwe.

Basic types

Basic types are entered by the user. Resolwe implements the backend handling (storage and retrieval) of basic types and GenBoard supports the HTML5 controls.

The following basic types are supported:

- `basic:boolean:` - boolean
- `basic:date:` - date (format *yyyy-mm-dd*)
- `basic:datetime:` - date and time (format *yyyy-mm-dd hh:mm:ss*)
- `basic:decimal:` - decimal number (*e.g., -123.345*)
- `basic:integer:` - whole number (*e.g., -123*)
- `basic:string:` - short string
- `basic:text:` - multi-line string
- `basic:url:link:` - visit link
- `basic:url:download:` - download link
- `basic:url:view:` - view link (in a popup or iframe)
- `basic:file:` - a file, stored on shared file system
- `basic:dir:` - a directory, stored on shared file system
- `basic:json:` - a JSON object, stored in MongoDB collection
- `basic:group:` - list of form fields (default if nothing specified)

The values of basic data types are different for each type, for example: `basic:file:` data type is a JSON dictionary: {"file": "file name"} `basic:dir:` data type is a JSON dictionary: {"dir": "directory name"} `basic:string:` data type is just a JSON string

Resolwe treats types differently. All but `basic:file:`, `basic:dir:` and `basic:json:` are treated as meta-data. `basic:file:` and `basic:dir:` objects are saved to the shared file storage, and `basic:json:` objects are stored in PostgreSQL bjson field. Meta-data entries have references to `basic:file:`, `basic:dir:` and `basic:json:` objects.

Data types

`Data` types are defined by processes. Each process is itself a `data:` sub-type named with the `type` attribute. A `data:` sub-type is defined by a list process outputs. All processes of the same `type` should have the same outputs.

`Data` type name:

- `data:<type>[:<sub-type>[...]]:`

1.3.4 The algorithm

Algorithm is the key component of a process. The algorithm transforms process's inputs into outputs. It is written as a sequence of Bash commands in process's `run.program` field.

Note: In this section, we assume that the program is written using the `bash` language and having the `expression-engine` requirement set to `jinja`.

To write the algorithm in a different language (*e.g.,* Python), just put it in a file with an appropriate *shebang* at the top (*e.g.,* `#!/usr/bin/env python2` for Python2 programs) and add it to the *tools* directory. To run it simply call the script with appropriate arguments.

For example, to compute a Volcano plot of the baySeq data, use:

```
volcanoplot.py diffexp_bayseq.tab
```

Platform utilities

Resolwe provides some convenience utilities for writing processes:

- `re-import`

  is a convenience utility that copies/downloads a file from the given temporary location, extracts/compresses it and moves it to the given final location. It takes six arguments:

  1. file's temporary location or URL
  2. file's final location
  3. file's input format, which can have one of the following forms:
     - `ending1|ending2`: matches files that end with `ending1` or `ending2` or a combination of `(ending1|ending2).(gz|bz2|zip|rar|7z|tgz|tar.gz|tar.bz2)`
     - `ending1|ending2|compression`: matches files that end with `ending1` or `ending2` or a combination of `(ending1|ending2).(gz|bz2|zip|rar|7z|tgz|tar.gz|tar.bz2)` or just with a supported compression format line ending `(gz|bz2|zip|rar|7z)`
  4. file's output format (*e.g.,* `fasta`)
  5. maximum progress at the end of transfer (a number between 0.0 and 1.0)
  6. file's output format, which can be one of the following:
     - `compress`: to produce a compressed file
     - `extract`: to produce an extracted file

     If this argument is not given, both, the compressed and the extracted file are produced.

For storing the results to process's output fields, Resolwe provides a series of utilities. They are described in the *Outputs* section.

Runtime

TODO: Write about BioLinux and what is available in the Docker runtime.

Inputs

To access values stored in process's input fields, use Jinja2's template language syntax for accessing variables. For example, to access the value of process's `fastq` input field, write `{{ fastq }}`.

In addition to all process's input fields, Resolwe provides the following system variables:

- `proc.case_ids`: ids of the corresponding cases
- `proc.data_id`: id of the data object
- `proc.data_dir`: file system path of the data object's directory
- `proc.slugs_path`: file system path to Resolwe's slugs

Resolwe also provides some custom built-in filters to access the fields of the referenced data objects:

- `id`: returns the id of the referenced data object
- `type`: returns the type of the referenced data object
- `name`: returns the value of the `static.name` field if it exists

For example, to use these filters on the `reads` field, use `{{ reads|id }}`, `{{ reads|type }}` or `{{ reads|name }}`, respectively.

You can also use any Jinja2's built in template tags and filters in your algorithm.

Note: All input variables should be considered *unsafe* and will be automatically quoted when used in your scripts. For example, the following call:

```
volcanoplot.py {{ reads.fastq.0.file }}
```

will actually be transformed into something like (depending on the value):

```
volcanoplot.py '/path/to/reads with spaces.gz'
```

If you do not want this behaviour for a certain variable and you are sure that it is safe to do so, you can use the `safe` filter as follows:

```
volcanoplot.py {{ known_good_input | safe }}
```

Outputs

Processes have three options for storing the results:

- as files in data object's directory (i.e. `{{ proc.data_dir }}`)
- as constants in process's output fields
- as entries in the MongoDB data storage

Note: Files are stored on a shared file system that supports fast read and write accesss by the processes. Accessing MongoDB from a process requires more time and is suggested for interactive data retrieval from GenPackages only.

Saving status

There are two special fields that you should use:

- `proc.rc`: the return code of the process
- `proc.progress`: the process's progress

If you set the `proc.rc` field to a positive value, the process will fail and its status will be set to `ERROR`. All processes that depend on this process will subsequently fail and their status will be set to `ERROR` as well.

The `proc.progress` field can be used to report processing progress interactively. You can set it to a value between 0 and 1 that represents an estimate for process's progress.

To set them, use the `re-progress` and `re-checkrc` utilities described in the *Saving constants* section.

Resolwe provides some specialized utilities for reporting process status:

- `re-error`

  takes one argument and stores it to `proc.error` field. For example:

  ```
  re-error "Error! Something went wrong."
  ```

- `re-warning`

  takes one argument and stores it to `proc.warning` field. For example:

  ```
  re-warning "Be careful there might be a problem."
  ```

- `re-info`

  takes one argument and stores it to `proc.info` field. For example:

  ```
  re-info "Just say hello."
  ```

- `re-progress`

  takes one argument and stores it to `proc.progress` field. The argument should be a float between 0 and 1 and represents an estimate for process's progress. For example, to estimate the progress to 42%, use:

  ```
  re-progress 0.42
  ```

- `re-checkrc`

  saves the return code of the previous command to `proc.rc` field. To use it, just call:

  ```
  re-checkrc
  ```

  As some programs exit with a non-zero return code, even though they finished successfully, you can pass additional return codes as arguments to the `re-checkrc` command and they will be translated to zero. For example:

  ```
  re-checkrc 2 15
  ```

  will set `proc.rc` to 0 if the return code is 0, 2 or 15, and to the actual return code otherwise.

  It is also possible to set the `proc.error` field with this command in case the return code is not zero (or is not given as one of the acceptable return codes). To do that, just pass the error message as the last argument to the `re-checkrc` command. For example:

```
re-checkrc "Error ocurred."

re-checkrc 2 "Return code was not 0 or 2."
```

Saving constants

To store a value in a process's output field, use the `re-save` utility. The `re-save` utility requires two arguments, a key (i.e. field's name) and a value (i.e. field's value).

For example, executing:

```
re-save quality_mean $QUALITY_MEAN
```

will store the value of the `QUALITY_MEAN` Bash variable in process's `quality_mean` field.

Note: To use the `re-save` utility, add `re-require common` to the beginning of the algorithm. For more details, see *Platform utilities*.

You can pass any JSON object as the second argument to the `re-save` utility, *e.g.*:

```
re-save foo '{"extra_output": "output.txt"}'
```

Note: Make sure to put the second argument into quotes (*e.g.,* "" or '') if you pass a JSON object containing a space to the `re-save` utility.

Saving files

A convinience function for saving files is:

```
re-save-file
```

It takes two arguments and stores the value of the second argument in the first argument's `file` subfield. For example:

```
re-save-file fastq $NAME.fastq.gz
```

stores `$NAME.fastq.gz` to the `fastq.file` field which has to be of type `basic:file:`.

To reference additional files/folders, pass them as extra arguments to the `re-save-file` utility. They will be saved to the `refs` subfield of type `basic:file:`. For example:

```
re-save-file fastq $NAME.fastq.gz fastqc/${NAME}_fastqc
```

stores `fastqc/${NAME}_fastqc` to the `fastq.refs` field in addition to storing `$NAME.fastq.gz` to the `fastq.file` field.

Note: Resolwe will automatically add files' sizes to the files' `size` subfields.

Warning: After the process has finished, Resolwe will automatically check if all the referenced files exist. If any file is missing, it will set the data object's status to `ERROR`. Files that are not referenced are automatically deleted by the platform, so make sure to reference all the files you want to keep!

Saving JSON blobs in MongoDB

To store a JSON blob to the MongoDB storage, simply create a field of type `data:json:` and use the `re-save` utility to store it. The platform will automatically detect that you are trying to store to a `data:json:` field and it will store the blob to a separate collection.

For example:

```
re-save etc { JSON blob }
```

will store the `{ JSON blob }` to the `etc` field.

Note: Printing a lot ot data to standard output can cause problems when using the Docker executor due to its current implementation. Therefore, it is advised to save big JSON blobs to a file and only pass the file name to the `re-save` function.

For example:

```
command_that_generates_large_json > json.txt
re-save etc json.txt
```

Warning: Do not store large JSON blobs into the data collection directly as this will slow down the retrieval of data objects.

1.4 API

The Resolwe framework provides a RESTful API through which most of its functionality is exposed.

TODO

1.4.1 Elasticsearch endpoints

Advanced lookups

All fields that can be filtered upon (as defined for each viewset) support specific lookup operators that can be used for some more advanced lookups.

Currently the supported lookup operators are:

- `lt` creates an ES range query with `lt` bound. Supported for number and date fields.
- `lte` creates an ES range query with `lte` bound. Supported for number and date fields.
- `gt` creates an ES range query with `gt` bound. Supported for number and date fields.
- `gte` creates an ES range query with `gte` bound. Supported for number and date fields.

- `in` creates an ES boolean query with all values passed as a should match. For GET requests, multiple values should be comma-separated.
- `exact` creates an ES query on the `raw` subfield of the given field, requiring the value to match exactly with the raw value that was supplied during indexing.

1.4.2 Limiting fields in responses

As responses from the Resolwe API can contain a lot of data, especially with nested JSON outputs and schemas, the API provides a way of limiting what is returned with each response.

This is achieved through the use of a special `fields` GET parameter, which can specify one or multiple field projections. Each projection defines what should be returned. As a working example, let's assume we have the following API response when no field projections are applied:

```
[
    {
        "foo": {
            "name": "Foo",
            "bar": {
                "level3": 42,
                "another": "hello"
            }
        },
        "name": "Boo"
    },
    {
        "foo": {
            "name": "Different",
        },
        "name": "Another"
    }
]
```

A field projection may reference any of the top-level fields. For example, by using the `fields=name` projection, we get the following result:

```
[
    {
        "name": "Boo"
    },
    {
        "name": "Another"
    }
]
```

Basically all fields not matching the projection are gone. We can go further and also project deeply nested fields, e.g., `fields=foo__name`:

```
[
    {
        "foo": {
            "name": "Foo"
        }
    },
    {
        "foo": {
```

```
            "name": "Different"
        }
    }
]
```

And at last, we can combine multiple projections by separating them with commas, e.g., `fields=name,foo__name`, giving us:

```
[
    {
        "foo": {
            "name": "Foo"
        },
        "name": "Boo"
    },
    {
        "foo": {
            "name": "Different"
        },
        "name": "Another"
    }
]
```

1.5 Type extension composition

Many types that are part of the core Resolwe framework contain logic that users of the framework may need to extend. To facilitate this in a controlled manner, the Resolwe framework provides a generic type extension composition system.

1.5.1 Making a type extendable

The composition system is very generic and as such can be used on any type. It provides a single method which allows you to retrieve a list of all registered extensions for a type or an instance of that type.

```
>>> composer.get_extensions(my_type_or_instance)
[<Extension1>, <Extension2>]
```

The type can then use this API to incorporate the registered extensions into its current instance however it chooses. Note that what these extensions are is entirely dependent upon the type that uses them.

For example, in the core Resolwe framework we make all index definitions extendable by using something like:

```
for extension in composer.get_extensions(attr):
    mapping = getattr(extension, 'mapping', {})
    index.mapping.update(mapping)
```

1.5.2 Writng an extension

On the other side, you can also define extensions for types that are using the above mentioned API. All extensions are automatically discovered during Django application registration if they are placed in a module called `extensions` in the given application.

Extensions can be registered using a simple API:

```
class MyExtension:
    pass

composer.add_extension('fully.qualified.type.Path', MyExtension)
```

Again, what the extension is depends on the type that is being extended. Now we describe some common extension types for types that are part of the Resolwe core.

Elasticsearch indices

It is possible to extend the `mapping` field of Elasticsearch indices by defining an extension as follows:

```
class ExtendedDataIndex:
    """Data ES index extensions."""
    mapping = {
        'source': 'output.source',
        'species': 'output.species',
        'build': 'output.build',
        'feature_type': 'output.feature_type',
    }

composer.add_extension('resolwe.flow.elastic_indexes.data.DataIndex',␣
↪ExtendedDataIndex)
```

Elasticsearch documents

It is possible to extend Elasticsearch documents using arbitrary fields by defining an extension as follows:

```
class ExtendedDataDocument:
    """Data ES document extensions."""
    source = dsl.Keyword()
    species = dsl.Text()
    build = dsl.Keyword()
    feature_type = dsl.Keyword()

composer.add_extension('resolwe.flow.elastic_indexes.data.DataDocument',␣
↪ExtendedDataDocument)
```

Data viewset

It is possible to extend the filters of the `Data` viewset by defining an extension as follows:

```
class ExtendedDataViewSet:
    """Data viewset extensions."""
    filtering_fields = ('source', 'species', 'build', 'feature_type')

    def text_filter(self, value):
        return [
            Q('match', species={'query': value, 'operator': 'and', 'boost': 2.0}),
            Q('match', source={'query': value, 'operator': 'and', 'boost': 2.0}),
            Q('match', build={'query': value, 'operator': 'and', 'boost': 2.0}),
            Q('match', feature_type={'query': value, 'operator': 'and', 'boost': 1.0}
↪),
```

```
        ]
composer.add_extension('resolwe.flow.views.data.DataViewSet', ExtendedDataViewSet)
```

1.6 Reference

1.6.1 Permissions shortcuts

resolwe.permissions.shortcuts.**_group_groups**(*perm_list*)

Group permissions by group.

Input is list of tuples of length 3, where each tuple is in following format:

```
(<group_id>, <group_name>, <single_permission>)
```

Permissions are regrouped and returned in such way that there is only one tuple for each group:

```
(<group_id>, <group_name>, [<first_permission>, <second_permission>,...])
```

Parameters **perm_list** (*list*) – list of touples of length 3

Returns list tuples with grouped permissions

Return type list

resolwe.permissions.shortcuts.**get_object_perms**(*obj*, *user=None*)

Return permissions for given object in Resolwe specific format.

Function returns permissions for given object `obj` in following format:

```
{
    "type": "group"/"user"/"public",
    "id": <group_or_user_id>,
    "name": <group_or_user_name>,
    "permissions": [<first_permission>, <second_permission>,...]
}
```

For `public` type `id` and `name` keys are omitted.

If `user` parameter is given, permissions are limited only to given user, groups he belongs to and public permissions.

Parameters

- **obj** (a subclass of *BaseModel*) – Resolwe's DB model's instance
- **user** (`User` or `None`) – Django user

Returns list of permissions object in described format

Return type list

1.6.2 Permissions utils

resolwe.permissions.utils.**copy_permissions**(*src_obj*, *dest_obj*)

Copy permissions form `src_obj` to `dest_obj`.

1.6.3 Flow Managers

Workflow workload managers.

resolwe.flow.managers.**manager**
The global manager instance.

Type *Manager*

Dispatcher

class resolwe.flow.managers.dispatcher.**Manager**(**args*, ***kwargs*)
The manager handles process job dispatching.

Each *Data* object that's still waiting to be resolved is dispatched to a concrete workload management system (such as Celery or SLURM). The specific manager for that system (descended from *BaseConnector*) then handles actual job setup and submission. The job itself is an executor invocation; the executor then in turn sets up a safe and well-defined environment within the workload manager's task in which the process is finally run.

communicate(*data_id=None*, *run_sync=False*, *save_settings=True*)
Scan database for resolving Data objects and process them.

This is submitted as a task to the manager's channel workers.

Parameters

- **data_id** – Optional id of Data object which (+ its children) should be processes. If it is not given, all resolving objects are processed.
- **run_sync** – If `True`, wait until all processes spawned from this point on have finished processing. If no processes are spawned, this results in a deadlock, since counts are handled on process finish.
- **save_settings** – If `True`, save the current Django settings context to the global state. This should never be `True` for "automatic" calls, such as from Django signals, which can be invoked from inappropriate contexts (such as in the listener). For user code, it should be left at the default value. The saved settings are in effect until the next such call.

discover_engines(*executor=None*)
Discover configured engines.

Parameters **executor** – Optional executor module override

execution_barrier()
Wait for executors to finish.

At least one must finish after this point to avoid a deadlock.

get_execution_engine(*name*)
Return an execution engine instance.

get_executor()
Return an executor instance.

get_expression_engine(*name*)
Return an expression engine instance.

handle_control_event(*message*)
Handle an event from the Channels layer.

Channels layer callback, do not call directly.

load_execution_engines(*engines*)
Load execution engines.

load_executor(*executor_name*)
Load process executor.

load_expression_engines(*engines*)
Load expression engines.

override_settings(***kwargs*)
Override global settings within the calling context.

Parameters **kwargs** – The settings overrides. Same use as for `django.test.override_settings()`.

reset(*keep_state=False*)
Reset the shared state and drain Django Channels.

Parameters **keep_state** – If `True`, do not reset the shared manager state (useful in tests, where the settings overrides need to be kept). Defaults to `False`.

run(*data*, *runtime_dir*, *argv*)
Select a concrete connector and run the process through it.

Parameters

- **data** – The *Data* object that is to be run.
- **runtime_dir** – The directory the executor is run from.
- **argv** – The argument vector used to spawn the executor.

class resolwe.flow.managers.dispatcher.**SettingsJSONifier**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Customized JSON encoder, coercing all unknown types into strings.

Needed due to the class hierarchy coming out of the database, which can't be serialized using the vanilla json encoder.

default(*o*)
Try default; otherwise, coerce the object into a string.

resolwe.flow.managers.dispatcher.**dependency_status**(*data*)
Return abstracted status of dependencies.

- `STATUS_ERROR` .. one dependency has error status or was deleted
- `STATUS_DONE` .. all dependencies have done status
- `None` .. other

Workload Connectors

The workload management system connectors are used as glue between the Resolwe Manager and various concrete workload management systems that might be used by it. Since the only functional requirement is job submission, they can be simple and nearly contextless.

Base Class

class resolwe.flow.managers.workload_connectors.base.**BaseConnector**
The abstract base class for workload manager connectors.

The main *Manager* instance in *manager* uses connectors to handle communication with concrete backend workload management systems, such as Celery and SLURM. The connectors need not worry about how jobs are discovered or how they're prepared for execution; this is all done by the manager.

submit(*data*, *runtime_dir*, *argv*)
Submit the job to the workload management system.

Parameters

- **data** – The *Data* object that is to be run.
- **runtime_dir** – The directory the executor is run from.
- **argv** – The argument vector used to spawn the executor.

Local Connector

class resolwe.flow.managers.workload_connectors.local.**Connector**
Local connector for job execution.

submit(*data*, *runtime_dir*, *argv*)
Run process locally.

For details, see *submit()*.

Celery Connector

class resolwe.flow.managers.workload_connectors.celery.**Connector**
Celery-based connector for job execution.

submit(*data*, *runtime_dir*, *argv*)
Run process.

For details, see *submit()*.

Slurm Connector

class resolwe.flow.managers.workload_connectors.slurm.**Connector**
Slurm-based connector for job execution.

submit(*data*, *runtime_dir*, *argv*)
Run process with SLURM.

For details, see *submit()*.

Listener

Standalone Redis client used as a contact point for executors.

class resolwe.flow.managers.listener.**ExecutorListener**(**args*, ***kwargs*)
The contact point implementation for executors.

check_critical_load()
Check for critical load and log an error if necessary.

clear_queue()
Reset the executor queue channel to an empty state.

handle_abort(*obj*)
Handle an incoming `Data` abort processing request.

Important: This only makes manager's state consistent and doesn't affect Data object in any way. Any changes to the Data must be applied over `handle_update` method.

Parameters **obj** – The Channels message object. Command object format:

```
{
    'command': 'abort',
    'data_id': [id of the :class:`~resolwe.flow.models.Data` object
                this command was triggered by],
}
```

handle_finish(*obj*)
Handle an incoming `Data` finished processing request.

Parameters **obj** – The Channels message object. Command object format:

```
{
    'command': 'finish',
    'data_id': [id of the :class:`~resolwe.flow.models.Data` object
                this command changes],
    'process_rc': [exit status of the processing]
    'spawn_processes': [optional; list of spawn dictionaries],
    'exported_files_mapper': [if spawn_processes present]
}
```

handle_log(*obj*)
Handle an incoming log processing request.

Parameters **obj** – The Channels message object. Command object format:

```
{
    'command': 'log',
    'message': [log message]
}
```

handle_update(*obj*, *internal_call=False*)
Handle an incoming `Data` object update request.

Parameters

- **obj** – The Channels message object. Command object format:

```
{
    'command': 'update',
    'data_id': [id of the :class:`~resolwe.flow.models.Data`
                object this command changes],
    'changeset': {
        [keys to be changed]
```

```
    }
}
```

- **internal_call** – If `True`, this is an internal delegate call, so a reply to the executor won't be sent.

hydrate_spawned_files(*exported_files_mapper*, *filename*, *data_id*)
Pop the given file's map from the exported files mapping.

Parameters

- **exported_files_mapper** – The dict of file mappings this process produced.
- **filename** – The filename to format and remove from the mapping.
- **data_id** – The id of the *Data()* object owning the mapping.

Returns The formatted mapping between the filename and temporary file path.

Return type dict

push_stats()
Push current stats to Redis.

run()
Run the main listener run loop.

Doesn't return until *terminate()* is called.

terminate()
Stop the standalone manager.

State

Synchronized singleton state container for the manager.

resolwe.flow.managers.state.**update_constants**()
Recreate channel name constants with changed settings.

This kludge is mostly needed due to the way Django settings are patched for testing and how modules need to be imported throughout the project. On import time, settings are not patched yet, but some of the code needs static values immediately. Updating functions such as this one are then needed to fix dummy values.

class resolwe.flow.managers.state.**ManagerState**(*key_prefix*)
State interface implementation.

This holds variables required to be shared between all manager workers and takes care of operation atomicity and synchronization. Redis facilitates storage shared between workers, whereas atomicity needs to be dealt with explicitly; this interface hides the Redis and Python details required to achieve syntax-transparent atomicity (such as being able to do `executor_count += 1`, a load-modify-store operation sequence).

Consumer

Manager Channels consumer.

class resolwe.flow.managers.consumer.**ManagerConsumer**(**args*, ***kwargs*)
Channels consumer for handling manager events.

control_event(*message*)
Forward control events to the manager dispatcher.

resolwe.flow.managers.consumer.**exit_consumer**()
Cause the synchronous consumer to exit cleanly.

resolwe.flow.managers.consumer.**run_consumer**(*timeout=None*, *dry_run=False*)
Run the consumer until it finishes processing.

Parameters

- **timeout** – Set maximum execution time before cancellation, or `None` (default) for unlimited.
- **dry_run** – If `True`, don't actually dispatch messages, just dequeue them. Defaults to `False`.

resolwe.flow.managers.consumer.**send_event**(*message*)
Construct a Channels event packet with the given message.

Parameters **message** – The message to send to the manager workers.

Utilities

Utilities for using global manager features.

resolwe.flow.managers.utils.**disable_auto_calls**()
Decorator/context manager which stops automatic manager calls.

When entered, automatic *communicate()* calls from the Django transaction signal are not done.

1.6.4 Flow Executors

Main standalone execution stub, used when the executor is run.

It should be run as a module with one argument: the relative module name of the concrete executor class to use. The current working directory should be where the `executors` module directory is, so that it can be imported with python's `-m <module>` interpreter option.

Usage format:

```
/path/to/python -m executors .executor_type
```

Concrete example, run from the directory where `./executors/` is:

```
/venv/bin/python -m executors .docker
```

using the python from the `venv` virtualenv.

Note: The startup code adds the concrete class name as needed, so that in the example above, what's actually instantiated is `.docker.run.FlowExecutor`.

Base Class

class resolwe.flow.executors.run.**BaseFlowExecutor**(**args*, ***kwargs*)
Represents a workflow executor.

end()
End process execution.

get_stdout()
Get process' standard output.

get_tools_paths()
Get tools paths.

run(*data_id*, *script*)
Execute the script and save results.

run_script(*script*)
Run process script.

start()
Start process execution.

terminate()
Terminate a running script.

update_data_status(***kwargs*)
Update (PATCH) Data object.

Parameters **kwargs** – The dictionary of `Data` attributes to be changed.

Flow Executor Preparer

Framework for the manager-resident executor preparation facilities.

class resolwe.flow.executors.prepare.**BaseFlowExecutorPreparer**
Represents the preparation functionality of the executor.

extend_settings(*data_id*, *files*, *secrets*)
Extend the settings the manager will serialize.

Parameters

- **data_id** – The `Data` object id being prepared for.
- **files** – The settings dictionary to be serialized. Keys are filenames, values are the objects that will be serialized into those files. Standard filenames are listed in `resolwe.flow.managers.protocol.ExecutorFiles`.
- **secrets** – Secret files dictionary describing additional secret file content that should be created and made available to processes with special permissions. Keys are filenames, values are the raw strings that should be written into those files.

get_environment_variables()
Return dict of environment variables that will be added to executor.

get_tools_paths()
Get tools' paths.

post_register_hook(*verbosity=1*)
Run hook after the 'register' management command finishes.

Subclasses may implement this hook to e.g. pull Docker images at this point. By default, it does nothing.

resolve_data_path(*data=None*, *filename=None*)
Resolve data path for use with the executor.

Parameters

- **data** – Data object instance
- **filename** – Filename to resolve

Returns Resolved filename, which can be used to access the given data file in programs executed using this executor

resolve_upload_path(*filename=None*)
Resolve upload path for use with the executor.

Parameters **filename** – Filename to resolve

Returns Resolved filename, which can be used to access the given uploaded file in programs executed using this executor

Docker Flow Executor

class resolwe.flow.executors.docker.run.**FlowExecutor**(**args*, ***kwargs*)
Docker executor.

end()
End process execution.

run_script(*script*)
Execute the script and save results.

start()
Start process execution.

terminate()
Terminate a running script.

Preparation

class resolwe.flow.executors.docker.prepare.**FlowExecutorPreparer**
Specialized manager assist for the docker executor.

get_environment_variables()
Return dict of environment variables that will be added to executor.

post_register_hook(*verbosity=1*)
Pull Docker images needed by processes after registering.

resolve_data_path(*data=None*, *filename=None*)
Resolve data path for use with the executor.

Parameters

- **data** – Data object instance
- **filename** – Filename to resolve

Returns Resolved filename, which can be used to access the given data file in programs executed using this executor

resolve_upload_path(*filename=None*)
Resolve upload path for use with the executor.

Parameters **filename** – Filename to resolve

Returns Resolved filename, which can be used to access the given uploaded file in programs executed using this executor

Local Flow Executor

class resolwe.flow.executors.local.run.**FlowExecutor**(**args*, ***kwargs*)
Local dataflow executor proxy.

Preparation

class resolwe.flow.executors.local.prepare.**FlowExecutorPreparer**
Specialized manager assist for the local executor.

extend_settings(*data_id*, *files*, *secrets*)
Prevent processes requiring access to secrets from being run.

Null Flow Executor

class resolwe.flow.executors.null.run.**FlowExecutor**(**args*, ***kwargs*)
Null dataflow executor proxy.

This executor is intended to be used in tests where you want to save the object to the database but don't need to run it.

1.6.5 Flow Models

Base Model

Base model for all other models.

class resolwe.flow.models.base.**BaseModel**(**args*, ***kwargs*)
Abstract model that includes common fields for other models.

class Meta
BaseModel Meta options.

contributor
user that created the entry

created
creation date and time

modified
modified date and time

name
object name

save(**args*, ***kwargs*)
Save the model.

slug
URL slug

version
process version

Collection Model

Postgres ORM model for the organization of collections.

class resolwe.flow.models.collection.**BaseCollection**(**args*, ***kwargs*)
Template for Postgres model for storing a collection.

class Meta
BaseCollection Meta options.

description
detailed description

descriptor
collection descriptor

descriptor_dirty
indicate whether *descriptor* doesn't match *descriptor_schema* (is dirty)

descriptor_schema
collection descriptor schema

save(**args*, ***kwargs*)
Perform descriptor validation and save object.

tags
tags for categorizing objects

class resolwe.flow.models.**Collection**(**args*, ***kwargs*)
Postgres model for storing a collection.

duplicate(*contributor=None*)
Duplicate (make a copy).

duplicated
duplication date and time

is_duplicate()
Return True if collection is a duplicate.

objects = <django.db.models.manager.ManagerFromCollectionQuerySet object>
manager

Data model

Postgres ORM model for keeping the data structured.

class resolwe.flow.models.**Data**(**args*, ***kwargs*)
Postgres model for storing data.

STATUS_DIRTY = 'DR'
data object is in dirty state

STATUS_DONE = 'OK'
data object is done

STATUS_ERROR = 'ER'
data object is in error state

STATUS_PROCESSING = 'PR'
data object is processing

STATUS_RESOLVING = 'RE'
data object is being resolved

STATUS_UPLOADING = 'UP'
data object is uploading

STATUS_WAITING = 'WT'
data object is waiting

checksum
checksum field calculated on inputs

collection
collection

delete(**args*, ***kwargs*)
Delete the data model.

descriptor
actual descriptor

descriptor_dirty
indicate whether *descriptor* doesn't match *descriptor_schema* (is dirty)

descriptor_schema
data descriptor schema

duplicate(*contributor=None*, *inherit_entity=False*, *inherit_collection=False*)
Duplicate (make a copy).

duplicated
duplication date and time

entity
entity

finished
process finished date date and time (set by `resolwe.flow.executors.run.BaseFlowExecutor.run` or its derivatives)

input
actual inputs used by the process

is_duplicate()
Return True if data object is a duplicate.

location
data location

named_by_user
track if user set the data name explicitly

objects = <django.db.models.manager.ManagerFromDataQuerySet object>
manager

output
actual outputs of the process

parents
dependencies between data objects

process
process used to compute the data object

process_cores
actual allocated cores

process_error
error log message

process_info
info log message

process_memory
actual allocated memory

process_pid
process id

process_progress
progress

process_rc
return code

process_warning
warning log message

resolve_secrets()
Retrieve handles for all basic:secret: fields on input.

The process must have the `secrets` resource requirement specified in order to access any secrets. Otherwise this method will raise a `PermissionDenied` exception.

Returns A dictionary of secrets where key is the secret handle and value is the secret value.

save(*render_name=False*, **args*, ***kwargs*)
Save the data model.

save_dependencies(*instance*, *schema*)
Save data: and list:data: references as parents.

save_storage(*instance*, *schema*)
Save basic:json values to a Storage collection.

scheduled
date and time when process was dispatched to the scheduling system (set by‘‘resolwe.flow.managers.dispatcher.Manager.run‘‘

size
total size of data's outputs in bytes

started
process started date and time (set by `resolwe.flow.executors.run.BaseFlowExecutor.run` or its derivatives)

status
Data status

It can be one of the following:

- *STATUS_UPLOADING*
- *STATUS_RESOLVING*
- *STATUS_WAITING*
- *STATUS_PROCESSING*

- *STATUS_DONE*
- *STATUS_ERROR*

tags
tags for categorizing objects

class resolwe.flow.models.**DataDependency**(**args*, ***kwargs*)
Dependency relation between data objects.

KIND_IO = 'io'
child uses parent's output as its input

KIND_SUBPROCESS = 'subprocess'
child was spawned by the parent

child
child data object

kind
kind of dependency

parent
parent data object

class resolwe.flow.models.**DataLocation**(**args*, ***kwargs*)
Location data of the data object.

get_path(*prefix=None*, *filename=None*)
Compose data location path.

get_runtime_path(*filename=None*)
Compose data runtime location path.

purged
indicate wether the object was processed by *purge*

subpath
subpath of data location

Entity–relationship model

Postgres ORM to define the entity–relationship model that describes how data objects are related in a specific domain.

class resolwe.flow.models.**Entity**(**args*, ***kwargs*)
Postgres model for storing entities.

collection
collection to which entity belongs

duplicate(*contributor=None*, *inherit_collection=False*)
Duplicate (make a copy).

duplicated
duplication date and time

is_duplicate()
Return True if entity is a duplicate.

move_to_collection(*source_collection*, *destination_collection*)
Move entity to destination collection.

objects = <django.db.models.manager.ManagerFromEntityQuerySet object>
manager

type
entity type

class resolwe.flow.models.**Relation**(**args*, ***kwargs*)
Relations between entities.

The Relation model defines the associations and dependencies between entities in a given collection:

```
{
    "collection": "<collection_id>",
    "type": "comparison",
    "category": "case-control study",
    "entities": [
        {"enetity": "<entity1_id>", "label": "control"},
        {"enetity": "<entity2_id>", "label": "case"},
        {"enetity": "<entity3_id>", "label": "case"}
    ]
}
```

Relation `type` defines a specific set of associations among entities. It can be something like `group`, `comparison` or `series`. The relation `type` is an instance of *RelationType* and should be defined in any Django app that uses relations (e.g., as a fixture). Multiple relations of the same type are allowed on the collection.

Relation `category` defines a specific use case. The relation category must be unique in a collection, so that users can distinguish between different relations. In the example above, we could add another `comparison` relation of `category`, say `Case-case study` to compare `<entity2>` with `<entity3>`.

Relation is linked to *resolwe.flow.models.Collection* to enable defining different relations structures in different collections. This also greatly speed up retrieving of relations, as they are envisioned to be mainly used on a collection level.

`unit` defines units used in partitions where it is applicable, e.g. in relations of type `series`.

category
category of the relation

collection
collection to which relation belongs

entities
partitions of entities in the relation

type
type of the relation

unit
unit used in the partitions' positions (where applicable, e.g. for serieses)

class resolwe.flow.models.**RelationType**(**args*, ***kwargs*)
Model for storing relation types.

name
relation type name

ordered
indicates if order of entities in relation is important or not

DescriptorSchema model

Postgres ORM model for storing descriptors.

class resolwe.flow.models.**DescriptorSchema**(**args*, ***kwargs*)
Postgres model for storing descriptors.

description
detailed description

schema
user descriptor schema represented as a JSON object

Process model

Postgres ORM model for storing processes.

class resolwe.flow.models.**Process**(**args*, ***kwargs*)
Postgres model for storing processes.

PERSISTENCE_CACHED = 'CAC'
cached persistence

PERSISTENCE_RAW = 'RAW'
raw persistence

PERSISTENCE_TEMP = 'TMP'
temp persistence

category
category

data_name
template for name of Data object created with Process

description
detailed description

entity_always_create
Create new entity, regardless of `entity_input` or `entity_descriptor_schema` fields.

entity_descriptor_schema
Slug of the descriptor schema assigned to the Entity created with *entity_type*.

entity_input
Limit the entity selection in *entity_type* to a single input.

entity_type
Automatically add *Data* object created with this process to an *Entity* object representing a data-flow. If all input `Data` objects belong to the same entity, add newly created `Data` object to it, otherwise create a new one.

get_resource_limits()
Get the core count and memory usage limits for this process.

Returns

A dictionary with the resource limits, containing the following keys:

- `memory`: Memory usage limit, in MB. Defaults to 4096 if not otherwise specified in the resource requirements.
- `cores`: Core count limit. Defaults to 1.

Return type dict

input_schema
process input schema (describes input parameters, form layout **"Inputs"** for *Data.input*)

Handling:

- schema defined by: *dev*
- default by: *user*
- changable by: *none*

is_active
designates whether this process should be treated as active

output_schema
process output schema (describes output JSON, form layout **"Results"** for *Data.output*)

Handling:

- schema defined by: *dev*
- default by: *dev*
- changable by: *dev*

Implicitly defined fields (by *resolwe.flow.management.commands.register()* or `resolwe.flow.executors.run.BaseFlowExecutor.run` or its derivatives):

- `progress` of type `basic:float` (from 0.0 to 1.0)
- `proc` of type `basic:group` containing:
  - `stdout` of type `basic:text`
  - `rc` of type `basic:integer`
  - `task` of type `basic:string` (celery task id)
  - `worker` of type `basic:string` (celery worker hostname)
  - `runtime` of type `basic:string` (runtime instance hostname)
  - `pid` of type `basic:integer` (process ID)

persistence
Persistence of *Data* objects created with this process. It can be one of the following:

- *PERSISTENCE_RAW*
- *PERSISTENCE_CACHED*
- *PERSISTENCE_TEMP*

Note: If persistence is set to `PERSISTENCE_CACHED` or `PERSISTENCE_TEMP`, the process must be idempotent.

---

requirements
process requirements

run
process command and environment description for internal use

Handling:

- schema defined by: *dev*
- default by: *dev*
- changable by: *dev*

scheduling_class
process scheduling class

type
data type

Storage model

Postgres ORM model for storing JSON.

class resolwe.flow.models.**Storage**(**args*, ***kwargs*)
Postgres model for storing storages.

data
corresponding data objects

json
actual JSON stored

objects = <resolwe.flow.models.storage.StorageManager object>
storage manager

Secret model

Postgres ORM model for storing secrets.

class resolwe.flow.models.**Secret**(**args*, ***kwargs*)
Postgres model for storing secrets.

ProcessMigrationHistory model

Postgres ORM model for storing proces migration history.

class resolwe.flow.models.**ProcessMigrationHistory**(**args*, ***kwargs*)
Model for storing process migration history.

DataMigrationHistory model

Postgres ORM model for storing data migration history.

class resolwe.flow.models.**DataMigrationHistory**(**args*, ***kwargs*)
Model for storing data migration history.

1.6.6 Flow Utilities

Data Purge

resolwe.flow.utils.purge.**get_purge_files**(*root*, *output*, *output_schema*, *descriptor*, *descriptor_schema*)
Get files to purge.

resolwe.flow.utils.purge.**location_purge**(*location_id*, *delete=False*, *verbosity=0*)
Print and conditionally delete files not referenced by meta data.

Parameters

- **location_id** – Id of the *DataLocation* model that data objects reference to.
- **delete** – If `True`, then delete unreferenced files.

resolwe.flow.utils.purge.**purge_all**(*delete=False*, *verbosity=0*)
Purge all data locations.

Resolwe Exceptions Utils

Utils functions for working with exceptions.

resolwe.flow.utils.exceptions.**resolwe_exception_handler**(*exc*, *context*)
Handle exceptions raised in API and make them nicer.

To enable this, you have to add it to the settings:

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'resolwe.flow.utils.exceptions.resolwe_exception_
↪handler',
}
```

Statistics

Various statistical utilities, used mostly for manager load tracking.

class resolwe.flow.utils.stats.**NumberSeriesShape**
Helper class for computing characteristics for numerical data.

Given a series of numerical data, the class will keep a record of the extremes seen, arithmetic mean and standard deviation.

to_dict()
Pack the stats computed into a dictionary.

update(*num*)
Update metrics with the new number.

class resolwe.flow.utils.stats.**SimpleLoadAvg**(*intervals*)
Helper class for a sort of load average based on event times.

Given a series of queue depth events, it will compute the average number of events for three different window lengths, emulating a form of 'load average'. The calculation itself is modelled after the Linux scheduler, with a 5-second sampling rate. Because we don't get consistent (time-wise) samples, the sample taken is the average of a simple moving window for the last 5 seconds; this is to avoid numerical errors if actual time deltas were used to compute the scaled decay.

add(*count*, *timestamp=None*)
Add a value at the specified time to the series.

Parameters

- **count** – The number of work items ready at the specified time.
- **timestamp** – The timestamp to add. Defaults to None, meaning current time. It should be strictly greater (newer) than the last added timestamp.

to_dict()
Pack the load averages into a nicely-keyed dictionary.

1.6.7 Flow Management

Delete Unreferenced Files

class resolwe.flow.management.commands.purge.**Command**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)
Purge files with no reference in Data objects, and orphaned storages.

add_arguments(*parser*)
Command arguments.

handle(**args*, ***options*)
Call *purge_all()*.

Register Processes

class resolwe.flow.management.commands.register.**Command**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)
Register processes.

add_arguments(*parser*)
Command arguments.

find_descriptor_schemas(*schema_file*)
Find descriptor schemas in given path.

find_schemas(*schema_path*, *schema_type='process'*, *verbosity=1*)
Find schemas in packages that match filters.

handle(**args*, ***options*)
Register processes.

register_descriptors(*descriptor_schemas*, *user*, *force=False*, *verbosity=1*)
Read and register descriptors.

register_processes(*process_schemas*, *user*, *force=False*, *verbosity=1*)
Read and register processors.

retire(*process_schemas*)
Retire obsolete processes.

Remove old process versions without data. Find processes that have been registered but do not exist in the code anymore, then:

- If they do not have data: remove them
- If they have data: flag them not active (`is_active=False`)

valid(*instance*, *schema*)
Validate schema.

1.6.8 Elastic

Framework for advanced indexing of Django models with ElasticSearch.

To register index processor, create *elastic_indexes.py* file int your app and put subclass of *BaseIndex* in it. It will automatically register and index all objects specified in it.

For building the index for the first time or manually updating it, run:

```
python manage.py elastic_index
```

Elastic Indices

Main two classes

class resolwe.elastic.indices.**BaseDocument**(*meta=None*, ***kwargs*)

Base document class to build ElasticSearch documents.

This is standard `elasticsearch-dsl` `DocType` class with already added fields for handling permissions.

groups_with_permissions = None

list of group ids with view permission on the object

public_permission = None

identifies if object has public view permission assigned

users_with_permissions = None

list of user ids with view permission on the object

class resolwe.elastic.indices.**BaseIndex**

Base index class.

Builds ElasticSearch index for specific type of objects. Index is based on document type defined in `document_type`. Fields are determined from document and are populated with one of the following methods (in the exact order):

- `get_<field_name>_value` method is used
- `mapping[<feild_name>]` is used - if value is callable, it is called with current object as only argument
- value is extracted from the object's field with the same name

To make the index, caall `run` function. Index is build for all objects in queryset. To build index for just one object, specify it in `obj` parameter of `run` function.

To work properly, subclass of this class must override following attributes:

- object_type - class to which object must belong to be processed
- document_class - subclass of *BaseDocument* that is used to build actual index

Additional (optional) methods and attributes that can be overriden are:

- mapping - mapping for transforming object into index
- *preprocess_object()*
- *filter()*

build(*obj=None*, *queryset=None*, *push=True*)

Build indexes.

connection_thread_id = None
id of thread id where connection was established

create_mapping()
Create the mappings in elasticsearch.

destroy()
Destroy an index.

document_class = None
document class used to create index

filter(*obj*)
Determine if object should be processed.

If `False` is returned, processingg of the current object will be aborted.

generate_id(*obj*)
Generate unique document id for ElasticSearch.

get_dependencies()
Return dependencies, which should trigger updates of this index.

get_object_id(*obj*)
Return unique identifier of the object.

Object's id is returned by default. This method can be overriden if object doesn't have `id` attribute.

get_permissions(*obj*)
Return users and groups with `view` permission on the current object.

Return a dict with two keys - `users` and `groups` - which contain list of ids of users/groups with `view` permission.

mapping = {}
mapping used for building document

object_type = None
type of object that are indexed, i.e. Django model

preprocess_object(*obj*)
Preprocess object before indexing.

This function is called before *func:process_object*. It can be used for advanced pre-processing of the object, i.e. adding annotations that will be used in multiple fields.

process_object(*obj*)
Process current object and push it to the ElasticSearch.

push()
Push built documents to ElasticSearch.

push_queue = None
list of built documents waiting to be pushed

queryset = None
queryset of objects to index

remove_object(*obj*)
Remove current object from the ElasticSearch.

search()
Return search query of document object.

testing_postfix = ''
auto generated ES index postfix used in tests

Elastic Viewsets

class resolwe.elastic.viewsets.**ElasticSearchMixin**(**args*, ***kwargs*)
Mixin to use Django REST Framework with ElasticSearch based querysets.

This mixin adds following methods:

- *order_search()*
- *filter_search()*
- *filter_permissions()*

filter_permissions(*search*)
Filter given query based on permissions of the user in the request.

Parameters **search** – ElasticSearch query object

filter_search(*search*)
Filter given search by the filter parameter given in request.

Parameters **search** – ElasticSearch query object

get_always_allowed_arguments()
Return query arguments which are always allowed.

get_query_param(*key*, *default=None*)
Get query parameter uniformly for GET and POST requests.

get_query_params()
Get combined query parameters (GET and POST).

order_search(*search*)
Order given search by the ordering parameter given in request.

Parameters **search** – ElasticSearch query object

Elastic Index Builder

Elastic Paginators

Paginator classes used in Elastic app.

class resolwe.elastic.pagination.**LimitOffsetPostPagination**
Limit/offset paginator.

This is standard limit/offset paginator from Django REST framework, with difference that it supports passing `limit` and `offset` attributes also in the body of the request (not just as query parameter).

Elastic Utils

Collection of convenient functions and shortcuts that simplifies using the app.

resolwe.elastic.utils.**const**(*con*)
Define a constant mapping for elastic search index.

This helper may be used to define index mappings, where the indexed value is always set to a specific constant. Example:

```
mapping = {'field': const('I am a constant')}
```

Elastic Management commands

Elastic app includes following Django management commands:

Command: elastic_index

Command: elastic_mapping

Command: elastic_purge

1.6.9 Resolwe Test Framework

Resolwe Test Cases

class resolwe.test.**TestCaseHelpers**(*methodName='runTest'*)

Mixin for test case helpers.

assertAlmostEqualGeneric(*actual*, *expected*, *msg=None*)

Assert almost equality for common types of objects.

This is the same as `assertEqual()`, but using `assertAlmostEqual()` when floats are encountered inside common containers (currently this includes `dict`, `list` and `tuple` types).

Parameters

- **actual** – object to compare
- **expected** – object to compare against
- **msg** – optional message printed on failures

keep_data(*mock_purge=True*)

Do not delete output files after tests.

setUp()

Prepare environment for test.

tearDown()

Cleanup environment.

class resolwe.test.**TransactionTestCase**(*methodName='runTest'*)

Base class for writing Resolwe tests not enclosed in a transaction.

It is based on Django's `TransactionTestCase`. Use it if you need to access the test's database from another thread/process.

setUp()

Initialize test data.

class resolwe.test.**TestCase**(*methodName='runTest'*)

Base class for writing Resolwe tests.

It is based on *TransactionTestCase* and Django's TestCase. The latter encloses the test code in a database transaction that is rolled back at the end of the test.

class resolwe.test.**ProcessTestCase**(*methodName='runTest'*)

Base class for writing process tests.

It is a subclass of *TransactionTestCase* with some specific functions used for testing processes.

To write a process test use standard Django's syntax for writing tests and follow the next steps:

1. Put input files (if any) in `tests/files` directory of a Django application.
2. Run the process using *run_process()*.
3. Check if the process has the expected status using *assertStatus()*.
4. Check process's output using *assertFields()*, *assertFile()*, *assertFileExists()*, *assertFiles()* and *assertJSON()*.

Note: When creating a test case for a custom Django application, subclass this class and over-ride the `self.files_path` with:

```
self.files_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'files
↪')
```

Danger: If output files don't exist in `tests/files` directory of a Django application, they are created automatically. But you have to check that they are correct before using them for further runs.

assertDir(*obj*, *field_path*, *fn*)

Compare process output directory to correct compressed directory.

Parameters

- **obj** (Data) – object that includes the directory to compare
- **field_path** (*str*) – path to *Data* object's field with the file name
- **fn** (*str*) – file name (and relative path) of the correct compressed directory to compare against. Path should be relative to the `tests/files` directory of a Django application. Compressed directory needs to be in `tar.gz` format.

assertDirExists(*obj*, *field_path*)

Assert that a directory in the output field of the given object exists.

Parameters

- **obj** – object that includes the file for which to check if it exists
- **field_path** – directory name/path

assertDirStructure(*obj*, *field_path*, *dir_struct*, *exact=True*)

Assert correct tree structure in output field of given object.

Only names of directories and files are asserted. Content of files is not compared.

Parameters

- **obj** (Data) – object that includes the directory to compare
- **dir_path** (*str*) – path to the directory to compare

- **dir_struct** (*dict*) – correct tree structure of the directory. Dictionary keys are directory and file names with the correct nested structure. Dictionary value associated with each directory is a new dictionary which lists the content of the directory. Dictionary value associated with each file name is `None`
- **exact** (*bool*) – if `True` tested directory structure must exactly match *dir_struct*. If `False` *dir_struct* must be a partial structure of the directory to compare

assertFields(*obj*, *path*, *value*)
Compare object's field to the given value.

The file size is ignored. Use assertFile to validate file contents.

Parameters

- **obj** (`Data`) – object with the field to compare
- **path** (*str*) – path to *Data* object's field
- **value** (*str*) – desired value of *Data* object's field

assertFile(*obj*, *field_path*, *fn*, ***kwargs*)
Compare a process's output file to the given correct file.

Parameters

- **obj** (`Data`) – object that includes the file to compare
- **field_path** (*str*) – path to *Data* object's field with the file name
- **fn** (*str*) – file name (and relative path) of the correct file to compare against. Path should be relative to the `tests/files` directory of a Django application.
- **compression** (*str*) – if not `None`, files will be uncompressed with the appropriate compression library before comparison. Currently supported compression formats are *gzip* and *zip*.
- **filter** (*FunctionType*) – function for filtering the contents of output files. It is used in `itertools.filterfalse()` function and takes one parameter, a line of the output file. If it returns `True`, the line is excluded from comparison of the two files.
- **sort** (*bool*) – if set to `True`, basic sort will be performed on file contents before computing hash value.

assertFileExists(*obj*, *field_path*)
Ensure a file in the given object's field exists.

Parameters

- **obj** (`Data`) – object that includes the file for which to check if it exists
- **field_path** (*str*) – path to *Data* object's field with the file name/path

assertFiles(*obj*, *field_path*, *fn_list*, ***kwargs*)
Compare a process's output file to the given correct file.

Parameters

- **obj** (`Data`) – object which includes the files to compare
- **field_path** (*str*) – path to *Data* object's field with the list of file names
- **fn_list** (*list*) – list of file names (and relative paths) of files to compare against. Paths should be relative to the `tests/files` directory of a Django application.

- **compression** (*str*) – if not `None`, files will be uncompressed with the appropriate compression library before comparison. Currently supported compression formats are *gzip* and *zip*.
- **filter** (*FunctionType*) – Function for filtering the contents of output files. It is used in `itertools.filterfalse` function and takes one parameter, a line of the output file. If it returns `True`, the line is excluded from comparison of the two files.
- **sort** (*bool*) – if set to `True`, basic sort will be performed on file contents before computing hash value.

assertFilesExist(*obj*, *field_path*)
Ensure files in the given object's field exists.

Parameters

- **obj** (`Data`) – object that includes list of files for which to check existance
- **field_path** (*str*) – path to *Data* object's field with the file name/path

assertJSON(*obj*, *storage*, *field_path*, *file_name*)
Compare JSON in Storage object to the given correct JSON.

Parameters

- **obj** (`Data`) – object to which the *Storage* object belongs
- **storage** (*Storage* or `str`) – object or id which contains JSON to compare
- **field_path** (*str*) – path to JSON subset in the *Storage*'s object to compare against. If it is empty, the entire object will be compared.
- **file_name** (*str*) – file name (and relative path) of the file with the correct JSON to compare against. Path should be relative to the `tests/files` directory of a Django application.

Note: The given JSON file should be compresed with *gzip* and have the `.gz` extension.

---

assertStatus(*obj*, *status*)
Check if object's status is equal to the given status.

Parameters

- **obj** (`Data`) – object for which to check the status
- **status** (*str*) – desired value of object's *status* attribute

files_path
Path to test files.

get_json(*file_name*, *storage*)
Return JSON saved in file and test JSON to compare it to.

The method returns a tuple of the saved JSON and the test JSON. In your test you should then compare the test JSON to the saved JSON that is commited to the repository.

The storage argument could be a Storage object, Storage ID or a Python dictionary. The test JSON is assigned a json field of the Storage object or the complete Python dictionary (if a dict is given).

If the file does not exist it is created, the test JSON is written to the new file and an exception is rased.

Parameters

- **file_name** (*str*) – file name (and relative path) of a JSON file. Path should be relative to the `tests/files` directory of a Django app. The file name must have a `.gz` extension.
- **storage** (*Storage*, `str` or `dict`) – Storage object, Storage ID or a dict.

Returns (reference JSON, test JSON)

Return type tuple

preparation_stage()
Context manager to mark input preparation stage.

run_process(*process_slug*, *input_={}*, *assert_status='OK'*, *descriptor=None*, *descriptor_schema=None*, *verbosity=0*, *tags=None*)
Run the specified process with the given inputs.

If input is a file, file path should be given relative to the `tests/files` directory of a Django application. If `assert_status` is given, check if *Data* object's status matches it after the process has finished.

Note: If you need to delay calling the manager, you must put the desired code in a `with transaction.atomic()` block.

Parameters

- **process_slug** (*str*) – slug of the *Process* to run
- **input_** (*dict*) – *Process*'s input parameters

Note: You don't have to specify parameters with defined default values.

- **assert_status** (*str*) – desired status of the *Data* object
- **descriptor** (*dict*) – descriptor to set on the *Data* object
- **descriptor_schema** (*dict*) – descriptor schema to set on the *Data* object
- **tags** (*list*) – list of tags that will be added to the created *Data* object

Returns object created by *Process*

Return type *Data*

run_processor(**args*, ***kwargs*)
Run process.

Deprecated method: use run_process.

setUp()
Initialize test data.

tearDown()
Clean up after the test.

class resolwe.test.**TransactionResolweAPITestCase**(*methodName='runTest'*)
Base class for testing Resolwe REST API.

This class is derived from Django REST Framework's APITransactionTestCase class and has implemented some basic features that makes testing Resolwe API easier. These features includes following functions:

_get_list (*user=None*, *query_params={}*)
Make `GET` request to `self.list_view` view.

If `user` is not `None`, the given user is authenticated before making the request.

Parameters **user** (`User` or `None`) – User to authenticate in request

Returns API response object

Return type Response

_get_detail (*pk*, *user=None*, *query_params={}*)
Make `GET` request to `self.detail_view` view.

If `user` is not `None`, the given user is authenticated before making the request.

Parameters

- **pk** (*int*) – Primary key of the coresponding object
- **user** (`User` or `None`) – User to authenticate in request

Returns API response object

Return type Response

_post (*data={}*, *user=None*, *query_params={}*)
Make `POST` request to `self.list_view` view.

If `user` is not `None`, the given user is authenticated before making the request.

Parameters

- **data** (*dict*) – data for posting in request's body
- **user** (`User` or `None`) – User to authenticate in request

Returns API response object

Return type Response

_patch (*pk*, *data={}*, *user=None*, *query_params={}*)
Make `PATCH` request to `self.detail_view` view.

If `user` is not `None`, the given user is authenticated before making the request.

Parameters

- **pk** (*int*) – Primary key of the coresponding object
- **data** (*dict*) – data for posting in request's body
- **user** (`User` or `None`) – User to authenticate in request

Returns API response object

Return type Response

_delete (*pk*, *user=None*, *query_params={}*)
Make `DELETE` request to `self.detail_view` view.

If `user` is not `None`, the given user is authenticated before making the request.

Parameters

- **pk** (*int*) – Primary key of the coresponding object
- **user** (`User` or `None`) – User to authenticate in request

Returns API response object

Return type Response

_detail_permissions(*pk*, *data={}*, *user=None*)
Make POST request to self.detail_view view.

If user is not None, the given user is authenticated before making the request.

Parameters

- **pk** (*int*) – Primary key of the coresponding object
- **data** (*dict*) – data for posting in request's body
- **user** (User or None) – User to authenticate in request

Returns API response object

Return type Response

It also has included 2 views made from referenced DRF's ViewSet. First mimic list view and has following links between request's methods and ViewSet's methods:

- GET -> list
- POST -> create

Second mimic detail view and has following links between request's methods and ViewSet's methods:

- GET -> retrieve
- PUT -> update
- PATCH -> partial_update
- DELETE -> destroy
- POST -> permissions

If any of the listed methods is not defined in the VievSet, corresponding link is omitted.

Note: self.viewset (instance of DRF's Viewset) and self.resource_name (string) must be defined before calling super setUp method to work properly.

---

self.factory is instance of DRF's APIRequestFactory.

assertKeys(*data*, *wanted*)
Assert dictionary keys.

detail_permissions(*pk*)
Get detail permissions url.

detail_url(*pk*)
Get detail url.

list_url
Get list url.

setUp()
Prepare data.

class resolwe.test.**ResolweAPITestCase**(*methodName='runTest'*)
Base class for writing Resolwe API tests.

It is based on *TransactionResolweAPITestCase* and Django's TestCase. The latter encloses the test code in a database transaction that is rolled back at the end of the test.

Resolwe Test Helpers and Decorators

resolwe.test.utils.**check_installed**(*command*)
Check if the given command is installed.

Parameters **command** (*str*) – name of the command

Returns (indicator of the availability of the command, message saying command is not available)

Return type tuple(bool, str)

resolwe.test.utils.**check_docker**()
Check if Docker is installed and working.

Returns (indicator of the availability of Docker, reason for unavailability)

Return type tuple(bool, str)

resolwe.test.utils.**with_custom_executor**(*wrapped=None*, ***custom_executor_settings*)
Decorate unit test to run processes with a custom executor.

Parameters **custom_executor_settings** (*dict*) – custom FLOW_EXECUTOR settings with which you wish to override the current settings

resolwe.test.utils.**with_docker_executor**(*wrapped=None*)
Decorate unit test to run processes with the Docker executor.

resolwe.test.utils.**with_null_executor**(*wrapper=None*, *enabled=None*, *adapter=None*)
Decorate unit test to run processes with the Null executor.

resolwe.test.utils.**with_resolwe_host**(*wrapper=None*, *enabled=None*, *adapter=None*)
Decorate unit test to give it access to a live Resolwe host.

Set RESOLWE_HOST_URL setting to the address where the testing live Resolwe host listens to.

Note: This decorator must be used with a (sub)class of LiveServerTestCase which starts a live Django server in the background.

---

resolwe.test.utils.**is_testing**()
Return current testing status.

This assumes that the Resolwe test runner is being used.

1.6.10 Resolwe Utilities

class resolwe.utils.**BraceMessage**(*fmt*, **args*, ***kwargs*)
Log messages with the new {}-string formatting syntax.

Note: When using this helper class, one pays no significant performance penalty since the actual formatting only happens when (and if) the logged message is actually outputted to a log by a handler.

Example of usage:

```
from resolwe.utils import BraceMessage as __

logger.error(__("Message with {0} {name}", 2, name="placeholders"))
```

Source: https://docs.python.org/3/howto/logging-cookbook.html#use-of-alternative-formatting-styles.

1.7 Resolwe Flow Design

The Resolwe Flow workflow engine comprises the execution framework and other layers which make up the internal data model and facilitate dependency resolution, permissions enforcement and process filtering.

1.7.1 Overview

Execution Framework

Flow consists of a number of active services which need to be running before job execution can proceed.

The core message transport and coordination facility, as currently used, is Redis. It serves as the central status hub for keeping track of shared dynamic information used by parts of the framework, and as a contact point for those parts of the framework that run remotely. These connect to well-known 'channels' (specially named Redis list objects), into which they can deposit JSON-formatted messages and commands.

Flow's execution manager, or just the 'manager', is an independent service which runs as a Django Channels event consumer. When objects are added to the database to be executed, they will trigger events for the appropriate channels. These will be processed by the manager, which will carry out all the preparatory tasks necessary to start execution and then commuicate with a concrete workload management system so that the job can eventually be scheduled and run on a worker node.

Finally, the jobs are executed by the aptly named 'executors'. These are run on worker nodes and act as local execution managers: preparing a controlled execution environment, running the job's code, collecting results and communicating them back to the manager which stores them in the database.

Utility Layers

Django's facilities are used for interfacing with the database, thus all models used in Flow are actually Django Model objects. The most important two models are the *Data* model and the *Process* model.

A Data object represents a single instance of data to be processed, i.e. a node in the flow graph being executed. It contains properties which mainly concern execution, such as various process and task IDs, output statuses and the results produced by executors.

A Process object represents the way in which its Data object will be 'executed', i.e. the type of node in the flow graph and the associated code. It contains properties defining its relationship to other nodes in the graph currently being executed, the permissions that control access rights for users and other processes, and the actual code that is run by the executors.

The code in the process object can be either final code that is already ready for execution, or it can be a form of template, for which an 'expression engine' is needed. An expression engine (the only one currently in use is Jinja) pre-processes the process' code to produce text that can then be executed by an 'execution engine'.

An execution engine is, simply put, the interpreter that will run the processed code, just after an additional metadata discovery step. It is done by the execution engine because the encoding might be language-dependent. The properties to be discovered include process resource limits, secret requirements, etc. These properties are passed on to the executor, so that it can properly set up the target environment. The only currently supported execution engine is Bash.

1.7.2 Technicalities

The Manager

Being a Django Channels consumer application, the Flow Manager is entirely event-driven and mostly contextless. The main input events are data object creation, processing termination and intermediate executor messages. Once run, it consists of two distinct servers and a modularized connection framework used to interface with workload managers used by the deployment site.

Dispatcher

This is the central job scheduler. On receipt of an appropriate event through Django Channels (in this service, only data object creation and processing termination), the dispatcher will scan the database for outstanding data objects. For each object found to still not be processed, dependencies will be calculated and scanned for completion. If all the requirements are satisfied, its execution cycle will commence. The manager-side of this cycle entails job preprocessing and a part of the environment preparation steps:

- The data object's process is loaded, its code preprocessed with the configured expression engine and the result of that fed into the selected execution engine to discover further details about the process' environemntal requirements (resource limits).
- The runtime directories on the global shared file system are prepared: file dependencies are copied out from the database, the process' processed code (as output by the expression engine) is stored into a file that the executor will run.
- The executor platform is created by copying the Flow Executor source code to the destination (per-data) directories on the shared file system, along with serialized (JSON) settings and support metadata (file lists, directory paths, Docker configuration and other information the configured executor will need for execution).
- After all this is done, control is handed over to the configured 'workload connector', see below for a description.

Listener

As the name might imply to some, the purpose of the listener is to listen for status updates and distressing thoughts sent by executors. The service itself is an independent (*i.e.* not Django Channels-based) process which waits for events to arrive on the executor contact point channels in Redis.

The events are JSON-formatted messages and include:

- processing status updates, such as execution progress and any computed output values,
- spawn commands, with which a process can request the creation of new data objects,
- execution termination, upon which the listener will finalize the Data object in question: delete temporary files from the global shared file system, update process exit code fields in the database, store the process' standard output and standard error sent by the executor and notify the dispatcher about the termination, so that any state internal to it may be updated properly,
- ancillary status updates from the executor, such as logging. Because executors are running remotely with respect to the manager's host machine, they may not have access to any centralized logging infrastructure, so the listener is used as a proxy.

Workload Connectors

Workload connectors are thin glue libraries which communicate with the concrete workload managers used on the deployment site. The dispatcher only contains logic to prepare execution environments and generate the command line necessary to kick off an executor instance. The purpose of the workload connector is to submit that command line

to the workload manager which will then execute it on one of its worker nodes. The currently supported workload managers are Celery, SLURM and a simple local dummy for test environments.

The Executor

The Flow Executor is the program that controls Process execution on a worker node managed by the site workload manager, for which it is a job. Depending on the configured executor, it further prepares an execution environment, configures runtime limitations enforced by the system and spawns the code in the Process object. The currently supported executor types are a simple local executor for testing deployments and a Docker-based one.

Once started, the executor will carry out any additional preparation based on its type (*e.g.* the Docker executor constructs a command line to create an instance of a pre-prepared Docker container, with all necessary file system mappings and communication conduits). After that, it executes the Process code as prepared by the manager, by running a command to start it (this need not be anything more complicated than a simple *subprocess.Popen*).

Following this, the executor acts as a proxy between the process and the database by relaying messages generated by the process to the manager-side listener. When the process is finished (or when it terminates abnormally), the executor will send a final cleanup message and terminate, thus finishing the job from the point of view of the workload manager.

1.7.3 Example Execution, from Start to Finish

- Flow services are started: the dispatcher Django Channels application and the listener process.
- The user, through any applicable intricacy, creates a Data object.
- Django signals will fire on creation and submit a data scan event to the dispatcher through Django Channels.
- The dispatcher will scan the database for outstanding data objects (alternatively, only for a specific one, given an ID). The following steps are then performed for each discovered data object whose dependencies are all processed:
- The runtime directory is populated with data files, executor source and configuration files.
- The process code template is run through an expression engine to transform it into executable text. This is also scanned with an execution engine to discover runtime resource limits and other process-local configuration.
- A command line is generated which can be run on a processing node to start an executor.
- The command line is handed over to a workload connector, which submits it as a job to the workload manager installed on the site.
- At this point, the dispatcher's job for this data object is done. Eventually, the workload manager will start processing the submitted job, thereby spawning an executor.
- The executor will prepare a safe runtime context, such as a Docker container, configure it with appropriate communication channels (stdin/out redirection or sockets) and run the command to execute the process code.
- The code executes, periodically generating status update messages. These are received by the executor and re-sent to the listener. The listener responds appropriately, updating database fields for the data object, notifying the dispatcher about lifetime events or forwarding log messages to any configured infrastructure.
- Once the process is done, the executor will send a finalizing command to the listener and terminate.
- The listener will notify the dispatcher about the termination and finalize the database status of this data object (processing exit code, outputs).
- The dispatcher will update processing states and statistics, and re-scan the database for data objects which might have dependencies on the one that just finished and could therefore potentially be started up.

1.8 Change Log

All notable changes to this project are documented in this file. This project adheres to Semantic Versioning.

1.8.1 Unreleased

Changed

- Add `username` to `current_user_permissions` field of objects on API
- Support retrieval of `Data.name` in Python process

1.8.2 20.1.0 - 2019-12-16

Added

- Add `description` field to Collection full-text search

1.8.3 20.0.0 - 2019-11-18

Changed

- **BACKWARD INCOMPATIBLE:** Remove `download` permission from Data objects, samples and collections and `add` permission from samples and collections
- **BACKWARD INCOMPATIBLE:** Remove `Entity.descriptor_completed` field

Fixed

- Fix Docker executor command with `--cpus` limit option. This solves the issue where process is killed before the timeout 30s is reached

1.8.4 19.1.0 - 2019-09-17

Added

- Support filtering by `process_slug` in `DataViewSet`

Fixed

- Fix `DictRelatedField` so it can be used in browsable-API
- Fix access to subfields of empty `GroupField` in Python processes

1.8.5 19.0.0 - 2019-08-20

Changed

- **BACKWARD INCOMPATIBLE:** Change relations between `Data`, `Entity` and `Collection` from `ManyToMany` to `ManyToOne`. In practice this means that `Data.entity`, `Data.colllecton` and `Entity.collection` are now `ForeignKey`-s. This also implies the following changes:
  - `CollectionViewSet` methods `add_data` and `remove_data` are removed
  - `EntityViewset` methods `add_data`,‘‘remove_data‘‘, `add_to_collection` and `remove_from_collection` are removed
  - `EntityQuerySet` and `Entity` method `duplicate` argument `inherit_collections` is renamed to `inherit_collection`.
  - `EntityFilter` FilterSet field `collections` is renamed to `collection`.
- **BACKWARD INCOMPATIBLE:** Change following fields in `DataSerializer`:
  - `process_slug`, `process_name`, `process_type`, `process_input_schema`, `process_output_schema` are removed and moved in `process` field which is now `DictRelatedField` that uses `ProcessSerializer` for representation
  - Remove `entity_names` and `collection_names` fields
  - add `entity` and `colection` fields which are `DictRelatedField`-s that use corresponding serializers for representation
  - Remove support for `hydrate_entities` and `hydrate_collections` query parameters
- **BACKWARD INCOMPATIBLE:** Remove `data` field in `EntitySerializer` and `CollectionSerializer`. This implies that parameter `hydrate_data` is no longer supported.
- **BACKWARD INCOMPATIBLE:** Remove `delete_content` paremeter in `delete` method of `EntityViewset` and `CollectionViewSet`. From now on, when `Entity`/`Collection` is deleted, all it’s objects are removed as well
- Gather all `Data` creation logic into `DataQuerySet.create` method

Added

- Enable sharing based on user email
- Support running tests with live Resolwe host on non-linux platforms
- Add `inherit_entity` and `inherit_collection` arguments to `Data.duplicate` and `DataQuerySet.duplicate` method
- Implement `DictRelatedField`

1.8.6 18.0.0 - 2019-07-15

Changed

- **BACKWARD INCOMPATIBLE:** Remove `parents` and `children` query filters from Data API endpoint

Added

- `/api/data/:id/parents` and `/api/data/:id/children` API endpoints for listing parents and children Data objects of the object with given `id`
- Add `entity_always_create` field to `Process` model

Fixed

- Make sure that Elasticsearch index exists before executing a search query

1.8.7 17.0.0 - 2019-06-17

Changed

- **BACKWARD INCOMPATIBLE:** Use Elasticsearch version 6.x
- **BACKWARD INCOMPATIBLE:** Bump `Django` requirement to version `2.2`
- **BACKWARD INCOMPATIBLE:** Remove not used `django-mathfilters` requirement

Added

- Support Python 3.7
- Support forward and reverse many-to-one relations in Elasticsearch
- Add `collection_names` field to `DataSerializer`
- Add test methods to `ProcessTestCase` that assert directory structure and content: `assertDirExists`, `assertDir`, and `assertDirStructure`
- Add `upload-dir` process

1.8.8 16.0.1 - 2019-04-29

Fixed

- Pin `django-priority-batch` to version `1.1` to fix compatibility issues

1.8.9 16.0.0 - 2019-04-16

Changed

- **BACKWARD INCOMPATIBLE:** Access to DataField members (in Python process input) changed from dict to Python objects. For example, `input_field.file_field['name']` changed to `input_field.file_field.path`.
- **BACKWARD INCOMPATIBLE:** Filters that are based on `django-filter` `FilterSet` now use dict-declaring-syntax. This requires that subclasses of respective filters modify their syntax too.
- Interactively save results in Python processes

Added

- Add get_data_id_by_slug method to Python processes' Process class
- Python process syntax enhancements:
  - Support `.entity_name` in data inputs
  - Easy access to process resources through `self.resources`
- Raise error if ViewSet receives invalid filter parameter(s)
- Report process error for exceptions in Python processes
- Report process error if spawning fails
- Automatically export files for spawned processes (in Python process syntax)
- Import files of Python process FileField inputs (usage: *inputs.src.import_file()*)

Fixed

- Interactively write to standard output within Python processes
- Fix writing to integer and float output fields
- Allow non-required `DataField` as Python process input

1.8.10 15.0.1 - 2019-03-19

Fixed

- Fix storage migration to use less memory

1.8.11 15.0.0 - 2019-03-19

Changed

- Log plumbum commands to standard output
- Change storage data relation from many-to-one to many-to-many
- Moved `purged` field from `Data` to `DataLocation` model

Added

- Add `run_process` method to `Process` to support triggering of a new process from the running Python process
- Add DataLocation model and pair it with Data model to handle data location
- Add `entity_names` field to `DataSerializer`
- Support duplication of `Data`, `Entity` and `Collection`
- Support moving entities between collections
- Support relations requirement in process syntax

1.8.12 14.4.0 - 2019-03-07

Changed

- Purge processes only not jet purged Data objects

Fixed

- Allow references to missing Data objects in the output of finished Data objects, as we don't have the control over what (and when) is deleted

1.8.13 14.3.0 - 2019-02-19

Added

- Add `scheduled` field to `Data` objects to store the date when object was dispatched to the scheduling system
- Add `purge` field to `Data` model that indicates whether `Data` object was processed by `purge`

Fixed

- Make Elasticsearch build arguments cache thread-safe and namespace cache keys to make sure they don't interfere
- Trigger the purge outside of the transaction, to make sure the Data object is commited in the database when purge worger grabs it

1.8.14 14.2.0 - 2019-01-28

Added

- Add `input` Jinja filter to access input fields

1.8.15 14.1.0 - 2019-01-17

Added

- Add `assertFilesExist` method to `ProcessTestCase`
- Add `clean_test_dir` management command that removes files created during testing

Fixed

- Support registration of Python processes inherited from `process.Process`
- Skip docker image pull if image exists locally. This solves the issue where pull would fail if process uses an image that is only used locally.

1.8.16 14.0.1 - 2018-12-17

Fixed

- Make sure that tmp dir exists in Docker executor

1.8.17 14.0.0 - 2018-12-17

Changed

- **BACKWARD INCOMPATIBLE:** Run data purge in a separate worker to make sure that listener replies to the executor within 60 seconds
- Use batcher for spawned processes in listener
- Increase Docker's memory limit for 100MB to make sure processes are not killed when using all available memory and tune Docker memory limits to avoid OOM.

Added

- Raise an exception in Docker executor if container doesn't start for 60 seconds
- Set `TMPDIR` environment variable in Docker executor to `.tmp` dir in data directory to prevent filling up container's local storage

Fixed

- Process SIGTERM signal in executor as expected - set the Data status to error and set the process_error field
- Clear cached Django settings from the manager's shared state on startup

1.8.18 13.3.0 - 2018-11-20

Changed

- Switch channels_redis dependency to upstream version

Added

- Python execution engine
- Support multiple entity types
- Support extending viewsets with custom filter methods
- Add *tags* attribute to `ProcessTestCase.run_process` method which adds listed tag to the created `Data` object
- Copy `Data` objects tags from parent objects for spawned `Data` objects and `Data` objects created by workflows

Fixed

- Fix manager shutdown in the test runner. If an unrecoverable exception occurred while running a test, and never got caught (e.g. an unpicklable exception in a parallel test worker), the listener would not get terminated properly, leading to a hang.
- Data and collection name API filters were fixed to work as expected (ngrams was switched to raw).

1.8.19 13.2.0 - 2018-10-23

Added

- Use prioritized batcher in listener

1.8.20 13.1.0 - 2018-10-19

Added

- Use batching for ES index builds

Fixed

- Fix handling of M2M dependencies in ES indexer

1.8.21 13.0.0 - 2018-10-10

Changed

- **BACKWARD INCOMPATIBLE:** Remove Data descriptors from Entity Elasticsearch index
- Support searching by `slug` and `descriptor_data` in entity viewset text search

Added

- Add tags to collections

1.8.22 12.0.0 - 2018-09-18

Changed

- **BACKWARD INCOMPATIBLE:** Switch `Collection` and `Entity` API viewsets to use Elasticsearch
- **BACKWARD INCOMPATIBLE:** Refactor `Relation` model, which includes:
  - renaming `position` to `partition`
  - renaming `label` to `category` and making it required
  - adding `unit`
  - making `collection` field required

  - requiring unique combination of `collection` and `category`
  - renaming partition's `position` to `label`
  - adding (integer) `position` to partition (used for sorting)
  - deleting `Relation` when the last `Entity` is removed
- **BACKWARD INCOMPATIBLE:** Remove rarely used parameters of the `register` command `--path` and `--schemas`.
- Omit `current_user_permissions` field in serialization if only a subset of fields is requested
- Allow slug to be null on update to enable slug auto-generation
- Retire obsolete processes. We have added the `is_active` field to the Process model. The field is read-only on the API and can only be changed through Django ORM. Inactive processes can not be executed. The `register` command was extended with the `--retire` flag that removes old process versions which do not have associated data. Then it finds the processes that have been registered but do not exist in the code anymore, and:
  - If they do not have data: removes them
  - If they have data: flags them not active (`is_active=False`)

Added

- Add support for URLs in `basic:file:` fields in Django tests
- Add `collections` and `entities` fields to Data serializer, with optional hydration using `hydrate_collections` and/or `hydrate_entities`
- Support importing large files from Google Drive in re-import
- Add `python3-plumbum` package to resolwe/base:ubuntu-18.04 image

Fixed

- Prevent mutation of `input_` parameter in `ProcessTestCase.run_process`
- Return 400 instead of 500 error when slug already exists
- Add trailing colon to process category default
- Increase stdout buffer size in the Docker executor

1.8.23 11.0.0 - 2018-08-13

Changed

- **BACKWARD INCOMPATIBLE:** Remove option to list all objects on Storage API endpoint
- Make the main executor non-blocking by using Python asyncio
- Debug logs are not send from executors to the listener anymore to limit the amount of traffic on Redis

Added

- Add size to Data serializer
- Implement `ResolweSlugRelatedField`. As a result, `DescriptorSchema` objects can only be referenced by `slug` (instead of `id`)
- Add options to filter by `type` and `scheduling_class` on Process API endpoint

Fixed

- Inherit collections from `Entity` when adding `Data` object to it

1.8.24 10.1.0 - 2018-07-16

Changed

- Lower the level of all `INFO` logs in elastic app to `DEBUG`

Added

- Add load tracking to the listener with log messages on overload
- Add job partition selection in the SLURM workload connector
- Add `slug` Jinja filter
- Set `Data` status to `ERROR` if executor is killed by the scheduling system

Fixed

- Include the manager in the documentation, make sure all references work and tidy the content up a bit

1.8.25 10.0.1 - 2018-07-07

Changed

- Convert the listener to use asyncio
- Switched to `channels_redis_persist` temporarily to mitigate connection storms

Fixed

- Attempt to reconnect to Redis in the listener in case of connection errors

1.8.26 10.0.0 - 2018-06-19

Changed

- **BACKWARD INCOMPATIBLE:** Drop support for Python 3.4 and 3.5
- **BACKWARD INCOMPATIBLE:** Start using Channels 2.x

Added

- Add the options to skip creating of fresh mapping after dropping ES indices with `elastic_purge` management command
- Add `dirname` and `relative_path` Jinja filters

1.8.27 9.0.0 - 2018-05-15

Changed

- Make sorting by contributor case insensitive in Elasticsearch endpoints
- Delete ES documents in post delete signal instead of pre delete one

Added

- **BACKWARD INCOMPATIBLE:** Add on-register validation of default values in process and schemas
- **BACKWARD INCOMPATIBLE:** Validate that field names in processes and schemas start with a letter and only contain alpha-numeric characters
- Support Python 3.6
- Add `range` parameter and related validation to fields of type `basic:integer:`, `basic:decimal`, `list:basic:integer:` and `list:basic:decimal`
- Support filtering and sorting by `process_type` parameter on Data API endpoint
- Add `dirname` Jinja filter
- Add `relative_path` Jinja filter

Fixed

- Add missing `list:basic:decimal` type to JSON schema
- Don't crash on empty `in` lookup
- Fix {{ requirements.resources.* }} variables in processes to take in to the account overrides specified in Django settings
- Create Elasticsearch mapping even if there is no document to push

1.8.28 8.0.0 - 2018-04-11

Changed

- **BACKWARD INCOMPATIBLE:** Use Elasticsearch version 5.x
- **BACKWARD INCOMPATIBLE:** Raise an error if an invalid query argument is used in Elasticsearch viewsets
- **BACKWARD INCOMPATIBLE:** Switch `Data` API viewset to use Elasticsearch
- Terminate the executor if listener response with error message
- `verbosity` setting is no longer propagated to the executor

- Only create Elasticsearch mappings on first push

Added

- Add `sort` argument to `assertFile` and `assertFiles` methods in `ProcessTestCase` to sort file lines before asserting the content
- Add `process_slug` field to `DataSerializer`
- Improve log messages in executor and workload connectors
- Add `process_memory` and `process_cores` fields to `Data` model and `DataSerializer`
- Support lookup expressions (`lt`, `lte`, `gt`, `gte`, `in`, `exact`) in ES viewsets
- Support for easier dynamic composition of type extensions
- Add `elastic_mapping` management command

Fixed

- Fix Elasticsearch index rebuilding after a dependant object is deleted
- Send response to executor even if data object was already deleted
- Correctly handle reverse m2m relations when processing ES index dependencies

1.8.29 7.0.0 - 2018-03-12

Changed

- **BACKWARD INCOMPATIBLE:** Remove Ubuntu 17.04 base Docker image due to end of lifetime
- **BACKWARD INCOMPATIBLE:** Remove support for Jinja in `DescriptorSchema`'s default values
- **BACKWARD INCOMPATIBLE:** Remove `CONTAINER_IMAGE` configuration option from the Docker executor; if no container image is specified for a process using the Docker executor, the same pre-defined default image is used (currently this is `resolwe/base:ubuntu-16.04`)
- Add mechanism to change test database name from the environment, appending a `_test` suffix to it; this replaces the static name used before

Added

- Add Ubuntu 17.10 and Ubuntu 18.04 base Docker images
- Add database migration operations for process schema migrations
- Add `delete_chunked` method to `Data` objects queryset which is needed due to Django's extreme memory usage when deleting a large count of `Data` objects
- Add `validate_process_types` utility function, which checks that all registered processes conform to their supertypes
- Add `FLOW_CONTAINER_VALIDATE_IMAGE` setting which can be used to validate container image names
- Only pull Docker images at most once per process in `list_docker_images`
- Add `FLOW_PROCESS_MAX_CORES` Django setting to limit the number of CPU cores used by a process

Fixed

- Make parallel test suite worker threads clean up after initialization failures
- Add mechanism to override the manager's control channel prefix from the environment
- Fix deletion of a `Data` objects which belongs to more than one `Entity`
- Hydrate paths in `refs` of `basic:file:`, `list:basic:file:`, `basic:dir:` and `list:basic:dir:` fields before processing `Data` object

1.8.30 6.1.0 - 2018-02-21

Changed

- Remove runtime directory during general data purge instead of immediately after each process finishes
- Only process the Data object (and its children) for which the dispatcher's `communicate()` was triggered
- Propagate logging messages from executors to the listener
- Use process' slug instead of data id when logging errors in listener
- Improve log messages in dispatcher

Added

- Add `descriptor_completed` field to the `Entity` filter
- Validate manager semaphors after each test case, to ease debugging of tests which execute processes

Fixed

- Don't set Data object's status to error if executor is run multiple times to mitigate the Celery issue of tasks being run multiple times
- Make management commands respect the set verbosity level

1.8.31 6.0.1 - 2018-01-29

Fixed

- Make manager more robust to ORM/database failures during data object processing
- Rebuild the ElasticSearch index after permission is removed from an object
- Trim `Data.process_error`, `Data.process_warning` and `Data.process_info` fields before saving them
- Make sure values in `Data.process_error`, `Data.process_warning` and `Data.process_info` cannot be overwritten
- Handle missing `Data` objects in `hydrate_input_references` function
- Make executor fail early when executed twice on the same data directory

1.8.32 6.0.0 - 2018-01-17

Changed

- **BACKWARD INCOMPATIBLE:** `FLOW_DOCKER_LIMIT_DEFAULTS` has been renamed to `FLOW_PROCESS_RESOURCE_DEFAULTS` and `FLOW_DOCKER_LIMIT_OVERRIDES` has been renamed to `FLOW_PROCESS_RESOURCE_OVERRIDES`
- **BACKWARD INCOMPATIBLE:** `Process.PERSISTENCE_TEMP` is not used for execution priority anymore
- **BACKWARD INCOMPATIBLE:** There is only one available manager class, which includes dispatch logic; custom manager support has been removed and their role subsumed into the new connector system
- **BACKWARD INCOMPATIBLE:** Removed `FLOW_DOCKER_MAPPINGS` in favor of new `FLOW_DOCKER_VOLUME_EXTRA_OPTIONS` and `FLOW_DOCKER_EXTRA_VOLUMES`
- Parent relations are kept even after the parent is deleted and are deleted when the child is deleted
- Dependency resolver in manager is sped up by use of parent relations
- Validation of `Data` inputs is performed on save instead of on create

Added

- Support for the SLURM workload manager
- Support for dispatching `Data` objects to different managers
- Support for passing secrets to processes in a controlled way using a newly defined `basic:secret` input type
- `is_testing` test helper function, which returns `True` when invoked in tests and `False` otherwise
- Add `collecttools` Django command for collecting tools' files in single location defined in `FLOW_TOOLS_ROOT` Django setting which is used for mapping tools in executor when `DEBUG` is set to `False` (but not in tests)

Fixed

- Fix `Data` object preparation race condition in `communicate()`
- Set correct executor in flow manager
- Make executors more robust to unhandled failures
- Calculate `Data.size` by summing `total_size` of all file-type outputs
- Don't change slug explicitly defined by user - raise an error instead
- Objects are locked while updated over API, so concurrent operations don't override each other
- Make manager more robust to unhandled failures during data object processing
- Fix manager deadlock during tests
- Fix ctypes cache clear during tests
- Don't raise `ChannelFull` error in manager's communicate call
- Don't trim predefined slugs in `ResolweSlugField`

1.8.33 5.1.0 - 2017-12-12

Added

- Database-side JSON projections for `Storage` models
- Compute total size (including refs size) for file-type outputs
- Add `size` field to `Data` model and migrate all existing objects

Change

- Change Test Runner's test directory creation so it always creates a subdirectory in `FLOW_EXECUTOR`'s `DATA_DIR`, `UPLOAD_DIR` and `RUNTIME_DIR` directories

Fixed

- Do not report additional failure when testing a tagged process errors or fails
- Fix Test Runner's `changes-only` mode when used together with a Git repository in detached `HEAD` state
- Fix handling of tags and test labels together in Test Runner's `changes-only` mode
- Fix parallel test execution where more test processes than databases were created during tests

1.8.34 5.0.0 - 2017-11-28

Changed

- **BACKWARD INCOMPATIBLE:** The `keep_data()` method in `TransactionTestCase` is no longer supported. Use the `--keep-data` option to the test runner instead.
- **BACKWARD INCOMPATIBLE:** Convert the manager to Django Channels
- **BACKWARD INCOMPATIBLE:** Refactor executors into standalone programs
- **BACKWARD INCOMPATIBLE:** Drop Python 2 support, require Python 3.4+
- Move common test environment preparation to `TestCaseHelpers` mixin

Fixed

- Fix parents/children filter on Data objects
- Correctly handle removed processes in the changes-only mode of the Resolwe test runner

1.8.35 4.0.0 - 2017-10-25

Added

- **BACKWARD INCOMPATIBLE:** Add option to build only subset of specified queryset in Elasticsearch index builder
- `--pull` option to the `list_docker_images` management command
- Test profiling and process tagging

- New test runner, which supports partial test suite execution based on changed files
- Add `all` and `any` Jinja filters

Changed

- **BACKWARD INCOMPATIBLE:** Bump Django requirement to version 1.11.x
- **BACKWARD INCOMPATIBLE:** Make `ProcessTestCase` non-transactional
- Pull Docker images after process registration is complete
- Generalize Jinja filters to accept lists of `Data` objects
- When new `Data` object is created, permissions are copied from collections and entity to which it belongs

Fixed

- Close schema (YAML) files after `register` command stops using them
- Close schema files used for validating JSON schemas after they are no longer used
- Close stdout used to retrieve process results in executor after the process is finished
- Remove unrelated permissions occasionally listed among group permissions on `permissions` endpoint
- Fix `ResolwePermissionsMixin` to work correctly with multi-words model names, i.e. `DescriptorSchema`
- Fix incorrect handling of offset/limit in Elasticsearch viewsets

1.8.36 3.1.0 - 2017-10-05

Added

- `resolwe/base` Docker image based on Ubuntu 17.04
- Support different dependency kinds between data objects

Fixed

- Serialize `current_user_permissions` field in a way that is compatible with DRF 3.6.4+
- API requests on single object endpoints are allowed to all users if object has appropriate public permissions

1.8.37 3.0.1 - 2017-09-15

Fixed

- Correctly relabel SELinux contexts on user/group files

1.8.38 3.0.0 - 2017-09-13

Added

- Add filtering by id on `descriptor_schma` API endpoint
- Support assigning descriptor schema by id (if set value is of type int) on `Collection`, `Data` and `Entity` endpoints
- `assertAlmostEqualGeneric` test case helper, which enables recursive comparison for almost equality of floats in nested containers

Changed

- **BACKWARD INCOMPATIBLE:** Run Docker containers as non-root user

Fixed

- Use per-process upload dir in tests to avoid race conditions

1.8.39 2.0.0 - 2017-08-24

Added

- `descriptor` jinja filter to get the descriptor (or part of it) in processes
- Ubuntu 14.04/16.04 based Docker images for Resolwe
- Add `list_docker_images` management command that lists all Docker images required by registered processes in either plain text or YAML
- Data status is set to `ERROR` and error message is appended to `process_error` if value of `basic:storage:` field is set to a file with invalid JSON

Changed

- **BACKWARD INCOMPATIBLE:** Quote all unsafe strings when evaluating expressions in Bash execution engine
- **BACKWARD INCOMPATIBILE:** Rename `permissions` attribute on API endpoints to `current_user_permissions`
- API `permissions` endpoint raises error if no owner is assigned to the object after applied changes
- `owner` permission cannot be assigned to a group
- Objects with public permissions are included in list API views for logged-in users
- Owner permission is assigned to the contributor of the processes and descriptor schemas in the `register` management command
- The base image Dockerfile is renamed to Dockerfile.fedora-26

Fixed

- Add `basic:url:link` field to the JSON schema
- Return more descriptive error if non-existing permission is sent to the `permissions` endpoint
- Handle errors occurred while processing Elasticsearch indices and log them
- Return 400 error with a descriptive message if permissions on API are assigned to a non-existing user/group

1.8.40 1.5.1 - 2017-07-20

Changed

- Add more descriptive message if user has no permission to add `Data` object to the collection when the object is created

Fixed

- Set contributor of `Data` object to public user if it is created by not authenticated user
- Remove remaining references to calling `pip` with `--process-dependency-links` argument

1.8.41 1.5.0 - 2017-07-04

Added

- Add Resolwe test framework
- Add `with_custom_executor` and `with_resolwe_host` test decorators
- Add `isort` linter to check order of imports
- Support basic test case based on Django's `TransactionTestCase`
- Support ES test case based on Django's `TransactionTestCase`
- Support process test case based on Resolwe's `TransactionTestCase`
- Add ability to set a custom command for the Docker executor via the `FLOW_DOCKER_COMMAND` setting.
- `get_url` jinja filter
- When running `register` management command, permissions are automatically granted based on the permissions of previous latest version of the process or descriptor schema.
- Set `parent` relation in spawned `Data` objects and workflows
- Relations between entities
- Resolwe toolkit Docker images
- Archive file process
- File upload processes
- Resolwe process tests
- Add `SET_ENV` setting to set environment variables in executor
- Support ordering by version for descriptor schema

- Add `NullExecutor`
- If `choices` are defined in JSON schema, value of field is validated with them
- Add cpu core, memory and network resource limits
- Add scheduling class for processes (`interactive`, `batch`), which replaces the previously unused process priority field
- Add `share_content` flag to the collection and entity permissions endpoint to also share the content of the coresponding object
- Add `delete_content` flag to the collection and entity destroy method on API to also delete the content of the coresponding object

Changed

- Support running tests in parallel
- Split `flow.models` module to multiple files
- Remove ability to set a custom executor command for any executor via the `FLOW_EXECUTOR['COMMAND']` setting.
- Rename `RESOLWE_API_HOST` setting and environment variable in executor to `RESOLWE_HOST_URL`
- Remove `keep_failed` function in tests.
- Rename `keep_all` function to `keep_data`.
- Manager is automatically run when new `Data` object is created
- Outputs of `Data` objects with status `Error` are not validated
- Superusers are no longer included in response in `permissions` endpoint of resources
- Remove `public_processes` field from the `Collection` model as it is never used
- Public users can create new `Data` objects with processes and descriptor schemas on which they have appropriate permissions
- Add custom `ResolweSlugField` and use it instead of `django-autoslug`

Fixed

- **SECURITY:** Prevent normal users from creating new `Processes` over API
- Configure parallel tests
- Isolate Elasticsearch indices for parallel tests
- Fix Docker container name for parallel tests
- Generate temporary names for upload files in tests
- Fix permissions in Elasticsearch tests
- Do not purge data in tests
- Remove primary keys before using cached schemas' in process tests
- Set appropriate SELinux labels when mounting tools in Docker containers
- `Data` objects created by the workflow inherit its permissions

- If user doesn't have permissions on the latest versions of processes and descriptor schemas, older ones are used or error is returned
- Support `data:` and `list:data:` types
- Set `Data` object status to error if worker cannot update the object in the database
- `Data` objects returned in `CollectionViewset` and `EntityViewset` are filtered by permissions of the user in request
- Public permissions are taken into account in elastic app
- Treat `None` field value as if the field is missing
- Copy parent's permissions to spawned `Data` objects

1.8.42 1.4.1 - 2017-01-27

Fixed

- Update instructions on preparing a release to no longer build the wheel distribution which currently fails to install Resolwe's dependency links

1.8.43 1.4.0 - 2017-01-26

Added

- Auto-process style, type tree and category index
- Support loading JSON from a file if the string passed to the `basic:json:` field is a file.
- `list:basic:integer:` field
- Data object's checksum is automatically calculated on save
- `get_or_create` end point for `Data` objects
- `basic:file:html:` field for HTML files
- Helper function for comparing JSON fields in tests
- Purge directories not belonging to any data objects
- Ordering options to API endpoints
- Workflow execution engine
- `data_by_slug` filter for jinja expression engine
- Export `RESOLWE_API_HOST` environment variable in executor
- Add `check_installed()` test utility function
- Add support for configuring the network mode of Docker executor
- Add `with_docker_executor` test utility decorator
- Support for Docker image requirements
- Support version in descriptor schema YAML files
- Add `Entity` model that allows grouping of `Data` objects
- Introduce priority of Data objects

- Data objects created with processes with temporary persistence are given high priority.
- Add `resolwe.elastic` application, a framework for advanced indexing of Django models with Elastic-Search

Changed

- Refactor linters, check PEP 8 and PEP 257
- Split expression engines into expression engines and execution engines
- Use Jinja2 instead of Django Template syntax
- Expression engine must be declared in `requirements`
- Set Docker Compose's project name to `resolwe` to avoid name clashes
- Expose `check_docker()` test utility function
- Update versionfield to 0.5.0
- Support Django 1.10 and update filters
- Executor is no longer serialized
- Put Data objects with high priority into `hipri` Celery queue.

Fixed

- Fix pylint warnings (PEP 8)
- Fix pydocstyle warnings (PEP 257)
- Take last version of process for spawned objects
- Use default values for descriptor fields that are not given
- Improve handling of validation errors
- Ignore file size in `assertFields`
- Order data objects in `CollectionViewSet`
- Fix tests for Django 1.10
- Add quotes to paths in a test process test-save-file

1.8.44 1.3.1 - 2016-07-27

Added

- Sphinx extension `autoprocess` for automatic process documentation

1.8.45 1.3.0 - 2016-07-27

Added

- Ability to pass certain information to the process running in the container via environment variables (currently, user's uid and gid)

- Explicitly set working directory inside the container to the mapped directory of the current `Data`'s directory
- Allow overriding any `FLOW_EXECUTOR` setting for testing
- Support GET request on /api/<model>/<id>/permissons/ url
- Add OWNER permissions
- Validate JSON fields before saving `Data` object
- Add basic:dir field
- `RESOLWE_CUSTOM_TOOLS_PATHS` setting to support custom paths for tools directories
- Add test coverage and track it with Codecov
- Implement data purge
- Add `process_fields.name` custom tamplate tag
- Return contributor information together with objects
- Added permissions filter to determine `Storage` permissions based on referenced `Data` object

Changed

- Move filters to separate file and systemize them
- Unify file loading in tests
- Simplify `ProcessTestCase` by removing the logic for handling different uid/gid of the user running inside the Docker container
- Upgrade to django-guardian 1.4.2
- Rename `FLOW_EXECUTOR['DATA_PATH']` setting to `FLOW_EXECUTOR['DATA_DIR']`
- Rename `FLOW_EXECUTOR['UPLOAD_PATH']` setting to `FLOW_EXECUTOR['UPLOAD_DIR']`
- Rename `proc.data_path` system variable to `proc.data_dir`
- Rename test project's data and upload directories to `.test_data` and `.test_upload`
- Serve permissions in new format
- Rename `assertFiles` method in `ProcessTestCase` to `assertFile` and add new `assertFiles` method to check `list:basic:file` field
- Make `flow.tests.run_process` function also handle file paths
- Use Travis CI to run the tests
- Include all necessary files for running the tests in source distribution
- Exclude tests from built/installed version of the package
- Put packaging tests in a separate Tox testing environment
- Put linters (pylint, pep8) into a separate Tox testing environment
- Drop django-jenkins package since we no longer use Jenkins for CI
- Move testing utilities from `resolwe.flow.tests` to `resolwe.flow.utils.test` and from `resolwe.permissions.tests.base` to `resolwe.permissions.utils.test`
- Add Tox testing environment for building documentation
- Extend Reference documentation

Fixed

- Spawn processors (add data to current collection)
- Set collection name to avoid warnings in test output
- Improve Python 3 compatibility
- Fix setting descriptor schema on create

1.8.46 1.2.1 - 2016-05-15

Added

- Add docker-compose configuration for PostgreSQL
- Processes can be created on API
- Enable spawned processes

Changed

- Move logic from `Collection` model to the `BaseCollection` abstract model and make it its parent
- Remove all logic for handling `flow_collection`
- Change default database user and port in test project's settings
- Keep track of upload files created during tests and purge them afterwards

Fixed

- Test processes location agnostic
- Test ignore timezone support

1.8.47 1.2.0 - 2016-05-06

Changed

- Rename `assertFileExist` to `assertFileExists`
- Drop `--process-dependency-links` from Tox's pip configuration
- Improve documentation on preparing a new release

Added

- Ability to use a custom executor command by specifying the `FLOW_EXECUTOR['COMMAND']` setting
- Make workload manager configurable in settings

Fixed

- Make Resolwe work with Python 3 again
- Fix tests
- Render data name again after inputs are resolved
- Ensure Tox installs the package from sdist
- Pass all Resolwe's environment variables to Tox's testing environment
- Ensure tests gracefully handle unavailability of Docker

1.8.48 1.1.0 - 2016-04-18

Changed

- Rename *process_register* manage.py command to *register*
- Reference process by slug when creating new Data object
- Run manager when new Data object is created through API
- Include full DescriptorSchema object when hydrating Data and Collection objects
- Add *djangorestframework-filters* package instead of *django-filters*

Added

- Tox tests for ensuring high-quality Python packaging
- Timezone support in executors
- Generating slugs with *django-autoslug* package
- Auto-generate Data name on creation based on template defined in Process
- Added endpoint for adding/removeing Data objects to/from Collection

Fixed

- Pass all Resolwe's environment variables to Tox's testing environment
- Include all source files and supplementary package data in sdist
- Make Celery engine work
- Add all permissions to creator of *flow_collection* Colection
- Set DescriptorSchema on creating Data objects and Collections
- Loading DescriptorSchema in tests
- Handle Exceptions if input field doesn't match input schema
- Trigger ORM signals on Data status updates
- Don't set status od Data object to error status if return code of tool is 0

1.8.49 1.0.0 - 2016-03-31

Changed

- Renamed Project to Collection
- Register processes from packages and custom paths
- Removed support for Python 3.3

Added

- Permissions
- API for flow
- Docker executor
- Expression engine support
- Celery engine
- Purge command
- Framework for testing processors
- Processor finders
- Support for Django 1.9
- Support for Python 3.5
- Initial migrations
- Introductory documentation

1.8.50 0.9.0 - 2015-04-09

Added

Initial release.

1.9 Contributing

1.9.1 Installing prerequisites

Make sure you have Python 3.6 installed on your system. If you don't have it yet, follow these instructions.

Resolwe requires PostgreSQL (9.4+). Many Linux distributions already include the required version of PostgreSQL (e.g. Fedora 22+, Debian 8+, Ubuntu 15.04+) and you can simply install it via distribution's package manager. Otherwise, follow these instructions.

The pip tool will install all Resolwe's dependencies from PyPI. Installing some (indirect) dependencies from PyPI will require having a C compiler (e.g. GCC) as well as Python development files installed on the system.

Note: The preferred way to install the C compiler and Python development files is to use your distribution's packages, if they exist. For example, on a Fedora/RHEL-based system, that would mean installing `gcc` and `python3-devel` packages.

1.9.2 Preparing environment

Fork the main Resolwe's git repository.

If you don't have Git installed on your system, follow these instructions.

Clone your fork (replace `<username>` with your GitHub account name) and change directory:

```
git clone https://github.com/<username>/resolwe.git
cd resolwe
```

Prepare Resolwe for development:

```
pip install -e .[docs,package,test]
```

Note: We recommend using pyvenv to create an isolated Python environment for Resolwe.

1.9.3 Preparing database

Create a `resolwe` database:

```
# Remove database if exists
dropdb resolwe

# Create database
createdb resolwe
```

Set-up database:

```
cd tests
./manage.py migrate
./manage.py createsuperuser --username admin --email admin@genialis.com
```

1.9.4 Registering processes

```
cd tests
./manage.py register
```

1.9.5 Running tests

To run the tests, use:

```
cd tests
./manage.py test resolwe --parallel=2
```

To run the tests with Tox, use:

```
tox -r
```

1.9.6 Building documentation

```
python setup.py build_sphinx
```

1.9.7 Submitting changes upstream

Signed commits are required in the Resolwe upstream repository. Generate your personal GPG key and configure Git to use it automatically.

1.9.8 Preparing release

Checkout the latest code and create a release branch:

```
git checkout master
git pull
git checkout -b release-<new-version>
```

Replace the *Unreleased* heading in `docs/CHANGELOG.rst` with the new version, followed by release's date (e.g. *13.2.0 - 2018-10-23*).

Note: Use Semantic versioning.

Commit changes to git:

```
git commit -a -m "Prepare release <new-version>"
```

Push changes to your fork and open a pull request:

```
git push --set-upstream <resolwe-fork-name> release-<new-version>
```

Wait for the tests to pass and the pull request to be approved. Merge the code to master:

```
git checkout master
git merge --ff-only release-<new-version>
git push <resolwe-upstream-name> master <new-version>
```

Tag the new release from the latest commit:

```
git checkout master
git tag -m "Version <new-version>" <new-version>
```

Note: Project's version will be automatically inferred from the git tag using setuptools_scm.

Push the tag to the main Resolwe's git repository:

```
git push <resolwe-upstream-name> master <new-version>
```

The tagged code will we be released to PyPI automatically. Inspect Travis logs of the Release step if errors occur.

Preparing pre-release

When preparing a pre-release (i.e. an alpha release), one can skip the "release" commit that updates the change log and just tag the desired commit with a pre-release tag (e.g. *13.3.0a1*). By pushing it to GitHub, the tagged code will be automatically tested by Travis CI and then released to PyPI.

Python Module Index

r

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

S

T

U

V

W