Developer guide

This section provides some more details on BIAFLOWS workflows and details how to compile and debug BIAFLOWS workflows Docker image locally and add them to an existing BIAFLOWS instance.

Details on Python wrapper script and JSON descriptor

The sequence of operations commonly performed by BIAFLOWS Python wrapper scripts is detailed in Table S4.1. A complete reference to BIAFLOWS workflows JSON descriptor can be found online.

Phase	Actions	Notes
Initialization*	Connect to BIAFLOWS Retrieve Problem Class Retrieve job parameters	
Prepare_data*	Create empty in_folder, out_folder, gt_folder, tmp_folder Download all images without _lbl suffix to in_folder Download all images with _lbl suffix to gt_folder Download all image file attachments to gt_folder	Folders are created in user home folder (gt = ground truth). File names: annotation files must have the same name as input images + *_attached*
Workflow call	Call workflow from command line and passing in_folder, out_folder and parameters	The images from in_folder are sequentially processed, the results are stored in out_folder
Upload_data*	Parse images from out_folder (typically binary masks) and for each image/slice create annotations (polygon or point) and export them to BIAFLOWS	(1) Plain objects: extract connected particles (2D/3D) from mask, create polygon contours (slice by slice) and set contour ID (color LUT) to mask object ID (2) Points: Find non null pixels/voxels. Create point annotation at this position (3) Skeletons: Project mask (fully or by block), dilate, find contour around skeleton
Upload_metrics*	For each input file: call ComputeMetrics passing pairs of out_file(s) / gt_file(s), problem class (string) and optional metric parameters. Export metrics keys/values to benchmark database	gt_file: same name as out_file If an attached file is expected (e.g. division text file), it is assumed at the same location as out_file / gt_file and with the same name as the image + *_attached*

Note: *can be skipped (depends on the flags passed to workflow container). For instance, for the local processing (no BIAFLOWS server) all steps are skipped while for a local BIAFLOWS instance upload_metrics may be skipped if the images to be processed are not annotated.

Figure S4.1. Typical steps of a BIAFLOWS Python wrapper script

Installing software required for development (only once)

As workflows run inside a Docker container and since their Python wrapper script interacts with a BIAFLOWS instance, it is required to install Docker and Python 3 on your local machine. Cytomine Python client is also required for development.

Docker installation instructions can be found here:

For Linux:
https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04

For Windows:
https://docs.docker.com/docker-for-windows/install/#install-docker-for-windows-desktop-app

Python 3 and Cytomine Python client instructions can be found here:
See https://doc.uliege.cytomine.org/display/ALGODOC/Data+access+using+Python+client
In the following steps, we will use the workflow "NucleiSegmentation-ImageJ" as reference:
https://github.com/Neubias-WG5/W_NucleiSegmentation-ImageJ

Step 1. **Uploading a new workflow descriptor to BIAFLOWS**

Workflows have first to be described through a JSON descriptor, e.g.:
https://github.com/Neubias-WG5/W_NucleiSegmentation-ImageJ/blob/master/descriptor.json

Currently, some sections have to be customized manually, and some conventions must be respected to allow automatic parsing by BIAFLOWS. We recommend using https://github.com/Neubias-WG5/W_Template/blob/master/descriptor.json as template for your JSON descriptor.

Choose a workflow name without space. The description field (supporting restricted HTML) should be filled to document the workflow and it will be displayed from BIAFLOWS UI.

As inputs (workflow parameters), the five parameters (cytomine_host, cytomine_public_key, cytomine_private_key, cytomine_id_project, cytomine_id_software) are mandatory.

The workflow parameters should also be described:

- id: the parameter name (e.g : "ij_radius")
- value-key: a reference for the parameter in the command line. Keep "@ID", which is a shorthand meaning "replace by the parameter id, in uppercase". In our example, it will be replaced at parsing time by "IJ_RADIUS"
- command-line-flag: At execution time, the value-key in the command line will be replaced by the command-line-flag followed by the parameter value. Keep "--@id". In our example, it will be replaced in the command line by "--ij_radius".
- name: a human readable name displayed in BIAFLOWS
- type: Number, String, Boolean
- optional: set to true only if the workflow execution is not influenced by the presence or the absence of the parameter (e.g a "verbose" parameter). Workflow parameters having an influence on the results should never be optional.
- default-value: the default value of the parameter (in BIAFLOWS interface).

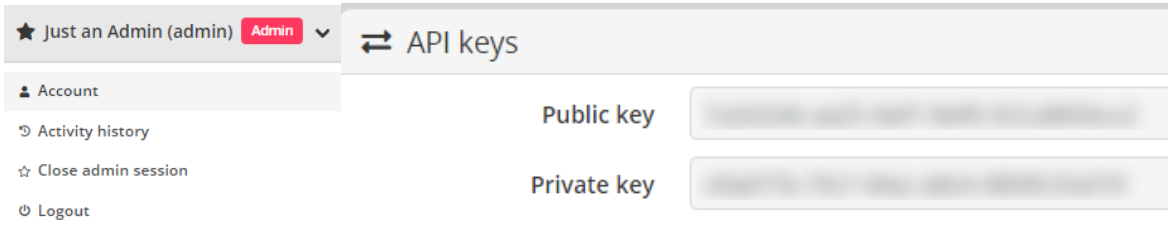
Do not forget to update the command line, with the parameter value keys. For instance, for workflow parameters ij_radius and ij_threshold:

```
python wrapper.py CYTOMINE_HOST CYTOMINE_PUBLIC_KEY CYTOMINE_PRIVATE_KEY
CYTOMINE_ID_PROJECT CYTOMINE_ID_SOFTWARE IJ_RADIUS IJ_THRESHOLD
```

To make a workflow available from a BIAFLOWS instance, it is currently required to publish its descriptor using Cytomine Python client. This can be performed by running the following Python code inside the folder holding the JSON descriptor you have created:

```
from cytomine import Cytomine
from cytomine.utilities.descriptor_reader import read_descriptor
with Cytomine(host, public_key, private_key) as c:
      read_descriptor("descriptor.json")
```

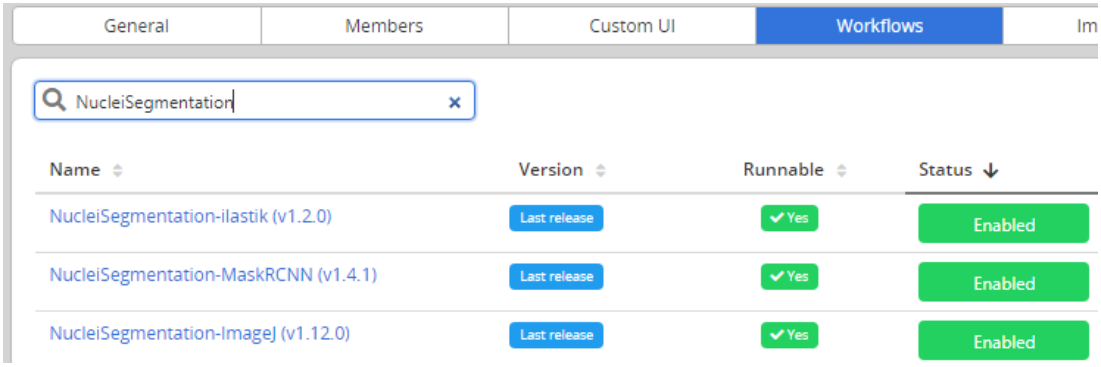
host is the url of your BIAFLOWS server, e.g. https://biaflows.neubias.org
public_key and ***private_key*** can be found from user Account page (section API KEYS)



Step 2. Linking a new workflow to a BIAFLOWS project

- From *Problems*, select the problem to which you want to add the workflow
- Go to *Problems* > *Configuration* > *Workflows* and enable the workflow

For now, as the workflow has been added manually, it will be referenced as **Not Runnable** and no version information will be provided from the UI.



Next, Go to *Projects* > *Configuration* and make sure that Jobs tab is activated (green)

Step 3. Creating the Dockerfile

Docker files specify the execution environment. They typically start by creating (FROM) a layer from an existing Docker image with basic operating system. Then they execute commands (RUN) to install specific software and libraries, and copy (ADD) files (e.g. the Python wrapper script and workflow script) into the execution environment the workflow will be called from. Finally, the ENTRYPOINT is set to the wrapper script.

A sample DockerFile is available here:
https://github.com/Neubias-WG5/W_NucleiSegmentation-ImageJ/blob/master/Dockerfile

If you do not know how to configure the Dockerfile, it is recommended to adapt the Dockerfile from an existing BIAFLOWS workflow using the same target software (e.g. an ImageJ macro).

Note: If you create a Dockerfile from scratch, always use the most accurate tag when referring to an existing Docker image (e.g. prefer python:3.6.9-stretch over python:3.6). If the tag is not accurate, the underlying docker image could change over time, heavily impairing reproducibility!

Step 4. Creating the wrapper script

It is recommended to adapt a wrapper script: 1) from same problem class, 2) processing image of same dimensionality (e.g. 3D), and 3) matching the software you are planning to use (e.g. ImageJ macro). In this case, only the workflow call (command line) needs to be adapted. A sample wrapper script is available here:
https://github.com/Neubias-WG5/W_NucleiSegmentation-ImageJ/blob/master/wrapper.py

Note: The flag **is_2d** should be used to specify if the images are strictly 2d or multidimensional.

Step 5. Building the workflow image, running it in a local container and debugging

A new workflow can be directly pushed to GitHub and be built in DockerHub, but it is preferable to test it locally beforehand. For this, it is required to build and run the Docker image locally:

Building the container (you need at least around 5GB disk space for this operation)

From a directory where you gathered the 4 files required to describe the workflow:

```
cd ~/Documents/Code/NEUBIAS/W_NucleiSegmentation-ImageJ$
sudo docker build -t seg2d .
```

Here `seg2d` is the name of the Docker image to build locally.

Running the Docker image:

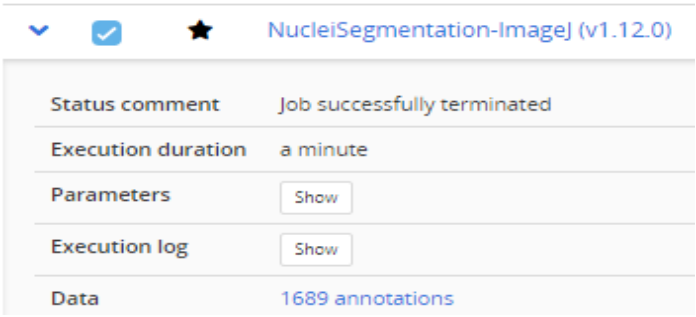
```
sudo docker run -it seg2d --host host --public_key public_key --private_key private_key --software_id software_id --project_id project_id --ij_threshold 15 --ij_radius 4
```

The list of command-line parameters should exactly match the parameters defined in the JSON descriptor file. BIAFLOWS instance URL and credentials should also be filled, as well as valid **workflow_id** (using **--software_id**) and **problem_id** (using **--projet_id**).

These IDs can be retrieved from the URL bar while respectively clicking on a problem (from BIAFLOWS **Problems** tab) and on a workflow (from BIAFLOWS **Workflows** tab):

In this example, workflow_id=23771763 and problem_id=5955.

If a workflow fails at execution this is reported in **Workflow runs** section. Some **Execution log** can be downloaded by expanding a workflow run from the blue arrow:



In that case, no associated benchmark metric is associated to this run. There is hence no risk that this would be left unnoticed by the user. For debugging, Docker can be run with an interactive session:

```
sudo docker run --entrypoint bash -it seg2d
```

If needed, it is also possible to launch the Docker with X enabled, e.g. to debug imageJ macro more easily:

```
xhost + sudo docker run --entrypoint bash -v /home/yourusername/tmp/test:/data -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -it seg2d
```

If you want to access local images without having to download them each time from BIAFLOWS, you can also attach a local folder to a folder inside the Docker container (-v option), for instance:

```
sudo docker run --entrypoint bash -v /home/yourusername/tmp/test:/data -
it seg2d
```

Some other useful Docker commands

Check if an image is running:	ps -a
Kill a running container:	sudo docker rm 65e88b2015df
Kill all running containers:	sudo docker rm $(sudo docker ps -a -q)
Download a specific container	sudo docker pull neubiaswg5/fiji-base:latest

Note: To download a recently updated workflow image, it is necessary to first remove older versions manually.

Step 6. Publishing a workflow with version control

Once your workflow is running properly, you can officially publish it with version control.

To allow automatic import to BIAFLOWS, the set of files previously described should be stored in a GitHub repository (linked to DockerHub) from an account trusted by the target BIAFLOWS instance.

The Github repository name **must be given by:**

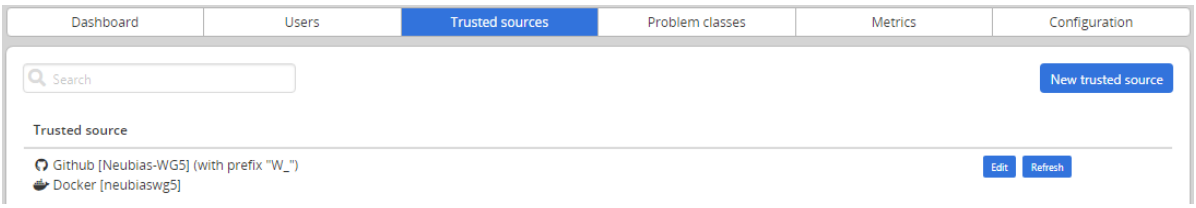
Github repo name = `{prefix}{workflow_name}`

where

- `{prefix}` is an optional prefix for the trusted source (see **Installing and populating BIAFLOWS locally**)
- `{workflow_name}` is the name of the workflow as given in the "name" field in the JSON descriptor (see Step 1).

For instance, for a trusted source with a prefix **W_**: **W_NucleiSegmentation-ImageJ**.

Adding/editing trusted sources is performed from Admin / Trusted sources (**Installing and populating BIAFLOWS locally**):



Step 7. Linking a GitHub repository to DockerHub (only once)

We assume that you created a trusted GitHub organization (e.g. **neubias-wg5**) and a workflow repository holding the 4 workflows files. It is now required to link DockerHub to