DiFlow

DATA
INTUITIVE

Graphical design and cover picture by Anneleen Maelfeyt (http://anneleenmaelfeyt.be/).

Typeset using LaTeX. More info to be found at http://www.data-intuitive.com.

Contents

Introduction

DiFlow[1] is an abstraction layer on top of NextFlow[2]'s DSL2[3]. DiFlow is a set of principles and guidelines for building NextFlow pipelines that allow the developer to declaratively define processing components and the user to declare the pipeline logic in a clean and intuitive way.

Viash[4] is a tool that (among other things) allows us to *use* DiFlow and make it practical, without the burden of maintaining boilerplate or *glue* code.

Functional Reactive Programming

FRP

If you're new to Functional Reactive Programming (FRP), here are a few pointers to posts and a video that introduce the concepts:

- An excellent Medium post[5] from Timo Stöttner
- The introduction[6] to Reactive Programming you've been missing from André Staltz.
- A very insightful presentation[7] by Staltz where he introduces FRP from first principles (with live coding).

In what follows, we will refer to *streams* in line with those authors but if you're used to working with Rx[8] you would call this an observable.

FRP in NextFlow

The `Channel`[9] class used by NextFlow, itself based on the DataFlow Programming Model[10] can in fact be regarded as an implementation of a Functional Reactive Programming library. Having said that, NextFlow allows one to to mix functional and imperative programming to the point that a developer is able to shoot its own foot.

Furthermore, `Channel`s can not be nested which complicates certain operations on the streams.

[1] https://pointer

[2] https://www.nextflow.io/

[3] https://www.nextflow.io/docs/latest/dsl2.html

[4] http://data-intuitive.com/viash_docs

[5] https://itnext.io/demystifying-functional-reactive-programming-67767dbe520b

[6] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

[7] https://www.youtube.com/watch?v=fdol03pcvMA

[8] http://reactivex.io/

[9] https://www.nextflow.io/docs/latest/channel.html

[10] https://en.wikipedia.org/wiki/Dataflow_programming

FRP for pipelines

NextFlow nor we are the first to understand that FRP is a good fit for pipeline development. Recent research and development also confirms this[11,12].

Abstraction

NextFlow DSL2

[11] https://soft.vub.ac.be/~mathsaey/skitter

[12] https://github.com/weng-lab/krews

Design Principles

Reproducibility

I originally did not include it as a design principle for the simple reason that I think it's obvious. This should be every researcher's top priority.

Pipeline Parameters vs Runtime Parameters

We make a strict distinction between parameters that are defined for the *FULL* pipeline and those that are defined at runtime.

Pipeline Parameters

We currently have 4 pipeline parameters: Docker prefix, `ddir`, `rdir` and `pdir`.

Runtime Parameters

Runtime parameters differ from pipeline parameters in that they may be different for parallel runs of a process. A few examples:

- Some samples may require different filter threshold than others
- After concatenation, clustering may be run with different cluster parameters
- etc.

In other words, it does not make sense to define those parameters for the full pipeline because they are not static.

In practice, we define the following as input of a module:

```
Channel( <Config Map>, <Sample ID or other unique ID>, <Input Path> )
```

The module returns a similar Channel:

```
Channel( <Updated Config Map>, <Sample ID>, <Output Path> )
```

The updated ConfigMap can be captured and written to disk as a log file. The idea is that it contains the full information of what has run, including the effective code.

Consistent API

Interchangeable components and component sets

Usage

Individual Components

Consider, e.g., Leiden. The following `platform_nextflow.yaml` was added:

```
type: nextflow
image: python-leiden
python:
  packages:
  - argparse
  - scanpy
  - python-igraph
  - leidenalg
  - hnswlib
workdir: /app
```

The image name is added as the `target_image` in the updated `platform_docker.yaml` in order to have a predictable target image after the (implicit) `docker build`.

In order to *test* this *module* using NXF, the following procedure can be followed:

1. Run Viash (version of 25/6/2020 with improved defaults for extensions):

```
viash export -f functionality.yaml -p platform_nexflow.yaml -o ../../../target/nxf/leiden
```

2. Run the (Dockerized) module with `---setup` such that the container is built.
3. Enter that directory and run (beware of the paths):

```
NXF_VER=20.04.1-edge nextflow run main.nf \
  --input ../../../src/cluster/leiden/test/pbmc_1k_protein_v3_filtered_feature_bc_matrix.norm.hvg.pca.nn.umap.h5ad \
  --output out/
```

The output is under `out/`.

Building the NXF modules

A script is available to generate the modules (at least for the components that contain a `platform_nextflow.yaml` file: `scripts/build_nxf_components.sh`.

In order to *use* the modules, the respective containers need to be available on the host. Those can be generated by issuing the script used for building the (Dockerized) components: `scripts/build_components.sh` as such:

```
scripts/build_components.sh ---setup
```

Caveats and Tips

Resources

When you run or export with the `DockerTarget`, resources are automatically added to the running container and stored under `/resources`. In case of the `NativeTarget`, this is not the case and since `NextFlowTarget` uses the `NativeTarget` it's the same there. That does not mean that resources specified in `functionality.yaml` is not available in these cases, we only have to point to them where appropriate.

The following snippet (from `ct/singler`) illustrates this:

```
par = list(
  input = "input.h5ad",
  output = "output.h5ad",
  reference = "HPCA",
  outputField = "cellType",
  pruningMADS = 3,
  outputFieldPruned = "celltype-pruned",
  reportOutputPath = "report.md"
)
## VIASH END
par$resources_dir <- resources_dir
```

In other words, `resources_dir` is automatically created by `viash` in all current 3 environments. This means that we can point to the `report.Rmd` file present in the resources like so:

```
rmarkdown::render(paste0(par$resources_dir, "/", "report.Rmd"), output_file = par$reportOutputPath)
```

Default values

In functionality, no option should have an empty string as value!

`target_image`

It makes sense to add the `target_image` attribute in the `docker_platform.yaml` file. This way, the resulting container image is predictable, rather than an autogenerated tag from `viash`.

Running the Docker setup

We don't have a solution yet for pre-generating the Docker images prior to starting a NXF pipeline. For the moment, we ask the user to run the build script for the Docker targets with the `---setup` option. This only works locally, it would for instance not work on a different (clean) node or in a Kubernetes cluster.

We are working on solutions or workarounds for this. Keep you posted!

Open issues

1. Multiple files as input for a component: E.g. the concat component uses multiple files to be joined. At the moment this does not seems to be possible.
2. Use of additional input files into a specific component. Some components do not only have input/output but require additional input. How should we map this?