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.

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

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.

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

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).

[5] https://itnext.io/demystifying-functional-reactive-programming-67767dbe520b
[6] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
[7] https://www.youtube.com/watch?v=fdol03pcvMA

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.

[8] http://reactivex.io/

FRP for pipelines

Other initiatives have consideren that FRP is a good fit for pipeline development. Recent research and development also confirms this[9,10].

[9] https://soft.vub.ac.be/~mathsaey/skitter/
[10] https://github.com/weng-lab/krews

NextFlow

FRP in NextFlow

The `Channel`[11] class used by NextFlow, itself based on the DataFlow Programming Model[12] 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.

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

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

NextFlow DSL(2)

DSL2[13] is a crucial development in NextFlow because it avoid having to maintain large, monolithic pipeline definitions in one file. With DSL2, developer can spin off functionality in separate files and `import` what is needed.

This also potentially opens up ways to build (reusable) modules that could be used in different projects. That is exactly what a lot of organizations need.

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

DiFlow

The NoPipeline approach

For developing the pipeline, we set out with a few goals in mind:

- Build modules where each modules deals with a specific (computational) task
- Make sure those modules can be reused
- Make sure the module functionality can be tested and validated
- Make sure modules have a consistent API, so that
  a. calling a module is straightforward
  b. including a module in a pipeline is transparent and seamless

Please note that nothing in these requirements has to do with running a pipeline itself. Rather, we consider this a bottom-up system whereby we first focus on a solid foundation before we actually start to tie things together.

That's why we call this the NoPipeline approach, similar to NoSQL where 'No' does not stand for *not*, but rather 'Not Only'. The idea is to focus on the pipeline aspect *after* the steps are properly defined and tested.

General Requirements

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.

Consistent API

When we started out with the project and chose to use NextFlow as a workflow engine, I kept on thinking that the level of abstraction should have been higher. With DSL1, all you could do was create one long list of NextFlow code, tied together by `Channel`s.

With DSL2, it became feasible to *organise* stuff in separate NextFlow files and import what is required. But in larger codebases, this is not really a benefit because every modules/workflow may have its own parameters and output. No structure is imposed. `Workflow`s are basically functions taking parameters in and returning values.

I think it makes sense to define an API and to stick to it as much as possible. This makes using the modules/workflows easier…

Flat Module Structure

We want to avoid having nested modules, but rather support a pool of modules to be mixed and matched.

As a consequence, this allows a very low threshold for including third-party modules: just add it to the collection of modules and import it in the pipeline. In order to facilitate the inclusion of such third-party modules that are developed in their own respective repositories, we added one additional layer in the hierarchy allowing for such a splitting.

Job Serialization

We avoid requiring the sources of the job available in the runtime environment, i.e., the Docker container. In other words, all code and config is serialized and sent with the *process*.

An abstract computation step

The module concept inspired us to think of an abstract way to represent a computation step and implement this in NextFlow. We wrote [Portash] to this end. But Portash had its shortcomings. The most important of which was that it did not adhere to separation of concerns: execution definition (what?) where mixed up with execution context (how?/where?). Moreover, dynamic nature of Portash lends itself well to running a tool as a service, but not so much in a batch process.

Nevertheless, we were able to express a generic NextFlow step as pure *configuration* that is passed to a process at runtime. This allows for some very interesting functionality. Some prototypes were developed, the last one of which could run a single-cell RNA pipeline from mapping to generating an integrated dataset combining different samples.

The run-configuration was provided by means of a Portash YAML spec residing in the module directory. It must be stressed that not requiring the component *code* to be already available inside the container is a big plus. It means a container contains dependencies, not the actual run script so the latter can be updated more frequently. This is especially useful during component and pipeline development.

Our first implementation had a few disadvantages:

- It contained a mix of what to run and how to run it, but it did not contain information on the container to run in. This had to be configured externally, but then the module is not an independent entity anymore.
- Specifying and overriding YAML content in Groovy is possible, but not something that is intuitive. We worked around that by letting the user specify custom configuration using a Groovy nested `Map`.
- The module functionality was abstracted with a consistent API and the difference between 2 modules was just a few lines of code with a different name or pointer. But still, one had to maintain that and making a similar change in a growing set of module files is a recipe for mistakes.

But overall, the concept of an abstract computation step proved to work, it was just that a few ingredients were still missing it seemed.

Appendix

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?