UPDF
WWW.UPDF.COM

Workflow writing basics

Table of Contents

- 1 Table of Contents
- 2 Introduction
- 3 Set up your IDE
  - 3.1 In pycharm
  - 3.2 In VScode
- 4 Submitting Workflows
- 5 Simple Workflows
  - 5.1 Minimal Workflow
  - 5.2 Templates
- 6 Workflow Parameters and Variables
  - 6.1 Static Parameters - Variables
  - 6.2 Dynamic Parameters
  - 6.3 Volumes and Dynamic Parameters
- 7 Loops, Conditionals and Parrallel execution
  - 7.1 Loops and Conditionals
  - 7.2 Parallel Executione
  - 7.3 Dynamic Parallel Execution
- 8 Artifacts
  - 8.1 Output Artifacts
  - 8.2 Input Artifacts
- 9 CronWorkflow and WorkflowTemplate
- 10 Final words

Introduction

So, you want to build a Workflow, hu? You've come to the right place.

Workflows are Argo's equivelent to Airflow's DAG, or Hkube's pipelines. It is basically a flow chart for your algorithms.
You can check a load of examples (from the internet) in this link. And you can check out my own (working) examples in this link.

Set up your IDE

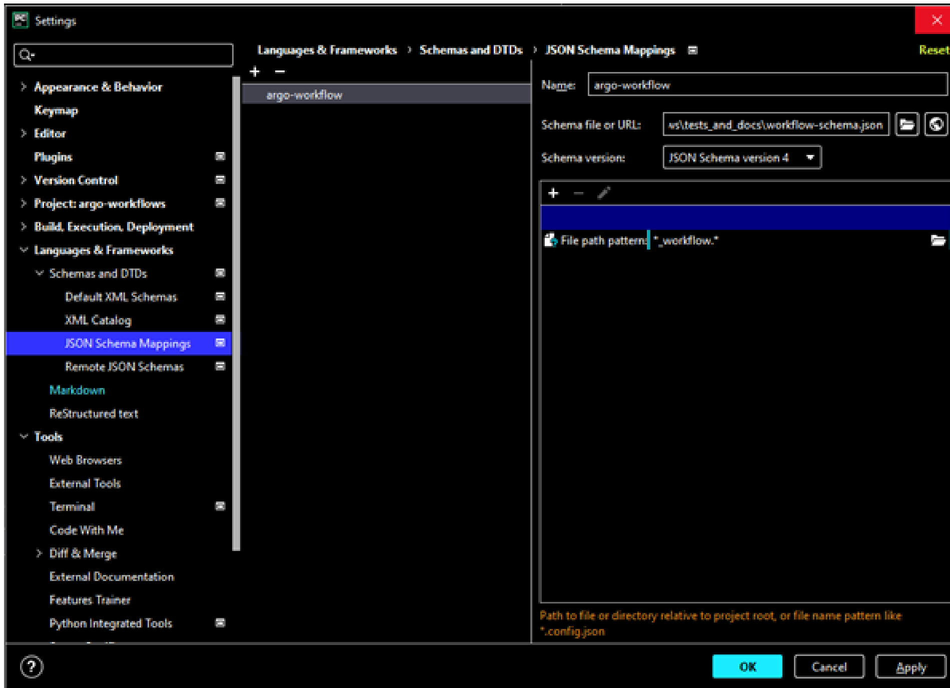
You can use the schema in this linkin order to set up your IDE so you'd have autocomplete for Argo's CRD's. It would help you build Workflow.

In pycharm

1. Download the *schema.json* .
2. In pycharm, go to: *Settings | Languages & Frameworks | Schemas and DTDs | JSON Schema Mappings*.
3. Define a map like this: press the plus, fill out a name and choose the schema file.

UPDF
WWW.UPDF.COM

4. Press on the little rightmost plus and choose a pattern for files that would have this autocomplete.



5. In the example above, every file with the endfix "_workflow." would be considerd as a workflow and we'd have autocomplete and syntax highlighting for it.
6. Hit ok.

In VScode

You need to do the same as in pycharm, but with the redhat-yaml extension. Click here to download it.

For a full guide - search on google "Argo set up your IDE" or something like "How to install json schemas for autocomplete vscode".

Submitting Workflows

- Argo API:

Submit your workflow in a POST with json containing your workflow to

<argo-workflows-host>/api/v1/workflows/<namespace>

In the HTTP POST request, your json needs to look like *{"workflow": <workflow as json>}.*

Note: If you **already built a workflow** and you want to activate it through the API with different input, make sure of the following:

- Your whole workflow is a *WorkflowTemplate*, so that you can submit to the api a workflow with a single *templateRef* step. The refrence should be to the entrypoint of your *WorkflowTemplate.*
- Your workflow takes it's initial input from the varible *{{workflow.parameters}}* .
- In your submitted workflow (in the http), you have the inputs in the field *spec.arguments.parameters*

- Openshift API:

Use *oc apply -f <workflow_name>.yaml.*

Notice: if you use openshift API you have to have a *metadata.name* field, *metadata.generateName* is not enough!

- Argo UI:

In the Workflow tab, click on "SUBMIT NEW WORKFLOW" on the top left.

Simple Workflows

UPDF
WWW.UPDF.COM

Minimal Workflow

A Workflow is basically an k8s\openshift resource, so you define it like any other resource - YAML or JSON files.
We write in YAML because it has comments and is human friendly (YAML for humans, JSON for computers).
Make sure to know some YAML basics (bonus: k8s YAML's) before diving in to this guide.

The most basic Workflow we can write is this:

Basic workflow

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow                     # new type of k8s spec
metadata:
  generateName: my-workflow-       # auto-generated name of the workflow spec
  name: my-workflow                # regular name
spec:
  entrypoint: my-template          # invoke the whalesay template
  templates:
  - name: my-template              # name of the template
    container:
      image: <registr/argo/alpine-python:3.11
      command: [echo]
      args: ["nadav four the win"]
```

Explainations:

- Lines 1-3: Regular k8s stuff. Notice that in *kind* you can write "Workflow", "WorkflowTemplate", "CronWorkflow" and more. These are **different types of Workflows**, all of them are simillar.
- Lines 4-5: Names to the workflow, you **have to have at least one of them**. *metadata.generateName* is preffered because Argo adds to it random characters (for example *my-workflow-n4tw.* The last characters are totally random).
  But if you use *oc apply* to a workflow it has to have *metadata.name.*
- Line 6: ***spec.entrypoint*** is the first *template* to run in this Workflow. **Templates** are like tiny "functions" that tell argo what pod to run, when to run it and how to run it. More on them later.
- Lines 9-13: Defines the first (and only) template. It runs the container with the *alpine-python* image, with the command and args *"echo nadav four the win".*
  Under *spec.templates[x].container* can be written good ol' fashioned k8s container. *x* is a number because ***spec.templates* is an array**.

Great! You can submit the Workflow through the submit section.

Now, what if you wanted to run two pods one after the other? Or maybe you even want to run two pods on the same time! Or even - god forbid - run three pods, one at the begining and two afterwards, simultameously?! D:
Fear not, Argo will help you fufill even your most notorious pod desiers.

Templates

As said above, templates are like tiny "functions" that are the building blocks of Workflows. They have multiple types:

- *Container*, as we've seen.
- *Steps*, dictates the order for other templates.
- *DAG,* simillar to steps.
- *Script*, allows you to run a script (python, node, bash...) on an image. Useful.
- *ContainerSet*, for multipule containers on a pod.
- *HTTP,* for http requests.

We don't need all of them. We will focus on *Container, Script,* and *Steps.*

So let's say we want to run two pods one after the other. We only have **one *entrypoint*,** so we can't make it run the *container* template (my-template) that we wrote. That will run that template only.
To prevent that, we will use the *steps* template (simillar to *dag*), like so:

Muilti-stepped workflow

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: steps-

spec:
```

UPDF
WWW.UPDF.COM

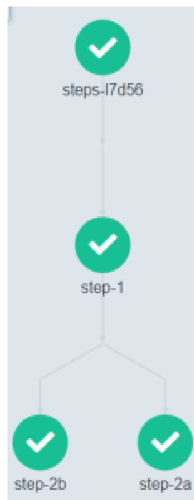
```
  entrypoint: my-steps

  # This spec contains three templates: my-steps, my-template, my-template-2
  templates:
  - name: my-steps
    # Instead of just running a container
    # This template has a sequence of steps
    steps:
    - - name: step-1            # step-1 is run before the following steps
        template: my-template

    - - name: step-2a           # double dash => run after previous step
        template: my-template

      - name: step-2b           # single dash => run in parallel with previous step
        template: my-template-2


  # This is the same template as from the previous example
  - name: my-template
    container:
      image: <registry>/argo/alpine-python:3.11
command: [echo]
      args: ["nadav for the win"]

  - name: my-template-2
    container:
      image: <registry>/argo/alpine-python:3.11
command: [echo]
      args: ["epstein didn't kill himself"]
```

Explainations:

- Lines 14-22: **Steps is an array of arrays.** It runs each second array simultaneously.
  So if, for example, we get: *steps=[[a], [b,c,d], [e,f]]*, we know that the pod *a* would run, then *b,c,d* together, and then *e,f* together.
  Each array is marked by a dash "-", so this array is actually *[["step-1"], ["step-2a", "step-2b"]].*
- Lines 26-36: We define two simple templates here that run according to the description in *steps.*

After submitting this example, we get:

<u>On the UI:</u>



<u>With the CLI:</u>

After running "*oc rsh <Argo-server pod name> argo get <workflow name>*":

UPDF
WWW.UPDF.COM

```
argo-workflow-argo-workflows-server-675df677dd-n5lgx argo get steps-17d56
Name:                steps-17d56
Namespace:
ServiceAccount:      unset (will run with the default ServiceAccount)
Status:              Succeeded
Conditions:
 PodRunning          False
 Completed           True
Created:             Tue Feb 22 12:01:58 +0000 (1 minute ago)
Started:             Tue Feb 22 12:01:58 +0000 (1 minute ago)
Finished:            Tue Feb 22 12:02:18 +0000 (1 minute ago)
Duration:            20 seconds
Progress:            3/3
ResourcesDuration:   9s*(1 cpu),9s*(100Mi memory)

STEP               TEMPLATE        PODNAME                     DURATION  MESSAGE
 ✔ steps-17d56     my-steps
 ├───✔ step-1      my-template     steps-17d56-13144749        9s
 └─┬─✔ step-2a     my-template     steps-17d56-1923859118      6s
   └─✔ step-2b     my-template-2   steps-17d56-1907081499      5s
```

Workflow Parameters and Variables

Static Parameters - Variables

In the previous section we learned how to structure a basic Workflow.
Notice that we have repetitive code over there (or repetitive YAML if you'd like) in lines 26-36. We wanted to echo two good and honest-to-god truthful statments, so we wrote a template for each of them.
But it's the same template, just with a different truthfull statement. So instead, we could write just one template, and insert the statments (the strings) as variables.

In order to do that, we'd write *"{{ variable }}"* or *"{{ =expression }}"* in the YAML. Then Argo would automatically replace these variables with the correct input.

Argo's variable substitution

Argo replaces the variables and expression during execution. So when a pod comes up with a variable, you can see in the YAML defenition of the pod the replaced variable.
That means that your variable values or inputs are written in the YAML file of the pods. Notice that in terms of security.

Also, it limits the length of the variables. The variables are strings that can't pass the few kb size.

Let's see how we can repeat the previous example with variables:

Muilti-stepped workflow

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: steps-

spec:
  entrypoint: my-steps

  # This spec contains three templates: my-steps, my-template
  templates:
  - name: my-steps
    steps:
    - - name: step-1            # step-1 is run before the following steps
        template: my-template
        arguments:
          parameters:
          - name: message
            value: "nadav for the win"

    - - name: step-2a           # double dash => run after previous step
        template: my-template
```

UPDF
WWW.UPDF.COM

```
        arguments:
          parameters:
          - name: message
            value: "nadav for the win"

      - name: step-2b           # single dash => run in parallel with previous step
        template: my-template   # changed my-template-2 to my-template
        arguments:
          parameters:
          - name: message
            value: "epstein didn't kill himself"


  - name: my-template
    inputs:
      parameters:
        - name: message
    container:
      image: <registry>/argo/alpine-python:3.11
command: [echo]
      args: ["{{inputs.parameters.message}}"]
```

Explainations:

- Lines 15-18: We define the step as before, but now we add **arguments**.
  There are two types of arguments: parameters and artifacts, I would explain artifacts in another time.
  Under parameters, we write a list of the parameters we want to pass to the template (like i've said, it's a mini function). We define a parameter with a *{"name": <name>, "value": "value"}* syntax.
- Lines 29-39: Another parameter, just this time it has a different value.
- Lines 36-48: If we want to use variables in our template, we have to define them!
  Here we define *inputs.parameters,* simillarly as we did in the first bullet. Notice that if we provide *value* here as well, it would be the **defualt value** of the variable.
- Line 42: This is how we use parameters in the workflow!
  In the YAML of the pod, it would look like this: *{"command":["echo"],"args":["nadav for the win"]}.*

Saved variables

Some variables are predefined by argo, they can be very helpful. For example:

- *{{workflow.name}}*
- *{{pod.name}}*
- *{{workflow.creationTimestamp}}*

For the full list search "Argo workflows variables" on google.

Dynamic Parameters

Dynamic parameters are parameters that are outputs from other pods that we want to forward. Their value is determined during runtime.
We would use them simillarly to the static parameters.

But first, we need to discuss how Argo reads outputs.
Each pod can have two types of outputs - artifacts and parameters. Artifacts are files that are saved and shared through an "Artifact repository" (usally s3-storage). But we will focus on parameters.

Argo knows to read the stdout of the pods and store that as variables. The stdout of a container is usally its logs, but argo utilizes them to pass parameters between pods.
So if, for example, your container counts my daily coffee intake in cups, and then prints their results in a json format: *{"result": 99}*, Argo **can read that "log" and you can load it as a variable**.

Let's see an example:

Simple stdout params output

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: example-params-
spec:
  entrypoint: steps
  templates:
```

UPDF
WWW.UPDF.COM

```
    name: steps
    steps:
    - - name: generate
        template: gen-random-int-python

    - - name: print
        template: print-params
        arguments:
          parameters:
          - name: res
            value: "{{steps.generate.outputs.result}}"


  - name: gen-random-int-python
    script:
      image: "<registry>/argo/alpine-python:3.11"
command: [python]
      source: |
        import random
        i = random.randint(420, 6969)
        print(i)

  - name: print-params
    inputs:
      parameters:
      - name: res

    script:
      image: "<registry>/argo/alpine-python:3.11"
command: [python]
      source: |
        res = {{inputs.parameters.res}}
        print(type(res), res*5)
```

Explainations:

- Lines 18: Now instead of writing a value, we use a parameter that will be set when the *generate* step will finish.
  This value says to Argo to take whatever *generate* had printed to its stdout (oc logs), and save it as the variable res.
  In this example it's a simple number, but we could also print a JSON file and transfer it.
- Lines 22-28: Here we use a ***script* template**.
  This template is simillar to *container*, but instead of running *command* with *args*, Argo creates a file with the string that is written in *script.source*, and runs that file.
  In this example we used the *script* template to execute python code that prints a random int in a legitimate and common range of numbers.
- Lines 39-40: In this python script we load the parameter.
  The parameter is written in the script exactly as it is read from the stdout of the previous pod. Take a look at the following notice:

Argo's parameter parsing

If line 28's printed *50* then in line 39 it would be written *res = 50* (picture it written in the YAML). And if line 28 printed *hello* then line 39 would be *res = hello*, which will **raise a python error** because hello is undefined.
To fix the following example, we need to make sure that **line 28 printed "*hello*" with quotations**, so that line 39 would be ***res = "hello",* which is valid python**.

Volumes and Dynamic Parameters

Let's say you're a good programmer (it's not a compliment, it's an assumption). And let's say that, being the good programmer that you are, your code prints pretty JSON logs to stdout.
So when you use *oc logs <pod-name>* or look at your pod in OpenShift UI, you want to see your logs. Maybe you even defined that your logs are sent directly from stdout to Splunk or Elastic (using annotations in OpenShift, see here).
But wait, you also want to use Argo and transfer parameters between pods in your Workflow! You can't have Argo read a parameter from stdout, it has all of your logs!
Fear not, youngling, for I shall teach you the ways of Argo.

We want to configure Argo so that it does'nt read your stdout and takes the parameters from there. Instead, it would be best if Argo just read a file with your output, preferably a JSON file.

WWW.UPDF.COM

So, you save all of your output in */some/path/output.json* . Now you need to tell Argo to read the parameter from this file.
But wait, Argo doesn't have access to your container, it only has access to its sidecar container (deafultly named *wait*, it's the *argo-exec* image).
So you need to **define a volume for your pod**, so that Argo's sidecar container could access your *output.json* file in that volume and read its contents.

Let's see look at a slightly more complex example:

Volumes and Parameters

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow

metadata:
  generateName: volume-outputs-example-
  name: volume-outputs-example

spec:
  podMetadata:
    annotations:
      collectord.io/index: my_splunk_index

  entrypoint: workflow-steps
  templates:
######################### steps template #################################
    - name: workflow-steps
      steps:
        - - name: generator
            template: generator
            arguments:
              parameters:
                - name: min
                  value: "420"
                - name: max
                  value: "6969"

        - - name: print-messege
            template: print-messege
            arguments:
              parameters:
                - name: number1
                  value: "{{steps.generator.outputs.parameters.number1}}"
                - name: number2
                  value: "{{steps.generator.outputs.parameters.number2}}"


######################### first template #################################
    - name: generator
      inputs:
        parameters:
          - name: min
          - name: max

      outputs:
        parameters:
          - name: number1
            valueFrom:
              path: /mnt/my_volume/number1.txt
          - name: number2
            valueFrom:
              path: /mnt/my_volume/number2.txt

      volumes:
        - name: my-volume
          emptyDir: {}


      script:
        image: "<registry>/argo/alpine-python:3.11"
command: [python]
        imagePullPolicy: Always
```

UPDF
WWW.UPDF.COM

```
        volumeMounts:
          - name: my-volume
            mountPath: /mnt/my_volume

        source: |
          from random import randrange

          range_min = {{ inputs.parameters.min }}
          range_max = {{ inputs.parameters.max }}
          random_number1 = randrange(range_min, range_max)
          random_number2 = randrange(range_min, range_max)

          with open("/mnt/my_volume/number1.txt", "w") as f:
            print(random_number1, file=f)
          with open("/mnt/my_volume/number2.txt", "w") as f:
            print(random_number2, file=f)

          print(f"Done! nums are {random_number1},{random_number2}")


######################### second template ################################
    - name: print-messege
      inputs:
        parameters:
          - name: number1
          - name: number2
      container:
        image: "<registry>/argo/alpine-python:3.11"
        command: [sh, -c]
        args: ["echo results are: {{inputs.parameters.number1}}, {{inputs.parameters.number2}}"]
```

Explainations:

- Lines 8-11: Metadata that will be written in the pods YAML. So this annotation will be added to the pod (the annotation sends logs from stdout to *my_splunk_index)*
- Lines 31-34: This is how we pass **defined outputs**.
  It's the same as before with the dynamic parameters, but now we want a specific output and not just *outputs.result* (which is stdout of the container).
  These outputs are defined in lines 44-51.
- Lines 44-51: Here we define the **template's outputs**. Notice that once again we can define *parameters* and *artifacts*, but we choose *parameters*.
  The names are matched to the names of the parameters above. The values are read from the files in the */mnt/my_volume/* folder.
  As before, the parameters will hold exactly what is written as a string in the files *number1.txt, number2.txt.*
- Lines 53-58: This is the **templates volume defenition**. Notice that in *mountPath* you choose the path of the folder that you want to share between the sidecar containers.
- Lines 73-76: In this python script, we save the results of our computation **inside the volume,** in the files */mnt/my_volume/number<x>.txt.*

Loops, Conditionals and Parrallel execution

You've learned the basics! You can almost call yourself a Junior Workflower (pls don't call yourself that).
Now it's time to move on to the fun stuff, and utilize the power of Argo-Workflows.

Loops and Conditionals

Templates are like mini-functions, so a Workflow is kind of a programming language.

Let's look at a neat example.
In the following Workflow, a coin is flipped. If it turns out heads - we won! If it turns out tails we lose.
Because we're petty, if the coin turns tails, we'll flip it again until we win (until it turns heads).
Let's see how to run a workflow in which we never lose:

Coin-Flip, loops and conditionals

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: coinflip-recursive
  generateName: coinflip-recursive-
```

UPDF
WWW.UPDF.COM

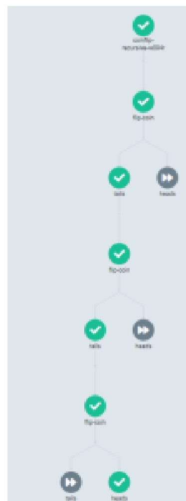
```
spec:
  entrypoint: steps-coinflip
  templates:
######################### steps template #################################
    - name: steps-coinflip
      steps:
        - - name: flip-coin
            template: flip-coin
        - - name: heads
            template: heads
            when: '{{steps.flip-coin.outputs.result}} == heads'
          - name: tails
            template: steps-coinflip
            when: '{{steps.flip-coin.outputs.result}} == tails'

######################### second template ##############################
    - name: flip-coin
      script:
        name: ''
        image: '<registry>/argo/alpine-python:3.11'
        command:
          - python
        source: |
          import random
          result = "heads" if random.randint(0,1) == 0 else "tails"
          print(result)
    - name: heads
      container:
        name: ''
        image: '<registry>/argo/alpine-python:3.11'
        command:
          - sh
          - '-c'
        args:
          - echo "it was heads"
```

Explainations:

- Lines 16: This is a ***conditional.*** This step will run only if this condition evulates to 'true'. For the full syntax of conditionals, search in google.
- Lines 18-19: If the conditional in line 17 will evulate to 'true', the *steps-coinflip* template will run.
  But wait! We're inside the *steps-coinflip* template! Argo will run this whole steps template once again - so the next pod that will run would be the *flip-coin.*
  Argo calls this ***Recursion***, although I prefer to call it a ***loop***. That is because it reminds more of a standard programming loop.

After submitting this example, we get:

On the UI:



With the CLI:

After running "oc rsh <Argo-server pod name> argo get <workflow name>":

```
$ oc rsh $POD_NAME argo get coinflip-recursive-w894r
Name:                coinflip-recursive-w894r
Namespace:           spectrum-np
ServiceAccount:      unset (will run with the default ServiceAccount)
Status:              Succeeded
Conditions:
 PodRunning          False
 Completed           True
Created:             Wed Feb 23 08:24:20 +0000 (4 hours ago)
Started:             Wed Feb 23 08:24:20 +0000 (4 hours ago)
Finished:            Wed Feb 23 08:25:22 +0000 (4 hours ago)
Duration:            1 minute 2 seconds
Progress:            4/4
ResourcesDuration:   11s*(100Mi memory),11s*(1 cpu)

STEP                          TEMPLATE   PODNAME                              DURATION  MESSAGE
 ✔ coinflip-recursive-w894r   coinflip
 ├───✔ flip-coin              flip-coin  coinflip-recursive-w894r-245495126   6s
 └─┬─○ heads                  heads                                                     when 'tails == heads' evaluated false
   └─✔ tails                  coinflip
     ├───✔ flip-coin          flip-coin  coinflip-recursive-w894r-2035964074  9s
     └─┬─○ heads              heads                                                     when 'tails == heads' evaluated false
       └─✔ tails              coinflip
         ├───✔ flip-coin      flip-coin  coinflip-recursive-w894r-3853324182  13s
         └─┬─✔ heads          heads      coinflip-recursive-w894r-1836388531  4s
           └─○ tails          coinflip                                                  when 'heads == tails' evaluated false

This workflow does not have security context set. You can run your workflow pods more securely by setting it.
Learn more at https://argoproj.github.io/argo-workflows/workflow-pod-security-context/
```

Parallel Executione

Let's say you want to run a simillar template, but for different inputs or different base image.
Take a look at the next example:

Parallelism

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: loops-param-arg-
spec:
  entrypoint: loop-param-arg-example
  arguments:
    parameters:
    - name: os-list                                     # a list of items
      value: |
        [
          { "image": "argo/alpine-python", "tag": "3.11" },
          { "image": "argo/bullseye-python", "tag": "3.11" },
          { "image": "jellyfish/rhel-python-3.6", "tag": "latest" }
        ]

  templates:
  ######################### steps template ###############################

  - name: loop-param-arg-example
    inputs:
      parameters:
      - name: os-list
    steps:
    - - name: test-linux
        template: cat-os-release
        arguments:
          parameters:
          - name: image
            value: "{{item.image}}"
          - name: tag
            value: "{{item.tag}}"
        withParam: "{{inputs.parameters.os-list}}"      # parameter specifies the list to iterate over

  ######################### first template ###############################
  - name: cat-os-release
    inputs:
      parameters:
      - name: image
```

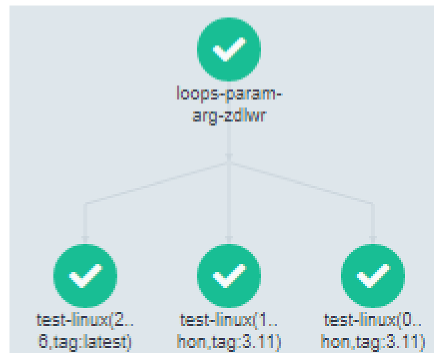
UPDF
WWW.UPDF.COM

```
      - name: tag
    container:
      image: "<registry>/{{inputs.parameters.image}}:{{inputs.parameters.tag}}"
command: [cat]
      args: [/etc/os-release]
```

Explainations:

- Lines 7-15: This is the way to enter paramters for the whole Workflow. At any place in the Workflow you can access these variables.
- Lines 29-33: The ***withParams*** key tells Argo to run this template once for every *item* in the input list (in parallel).
  The input is then called *item*, so in the first run: *item.image=argo/alpine-python.*

And the output:



Dynamic Parallel Execution

As of now, you're probably a clever Workflower. Or a determined reader. Or you just jumped to this part because your boss told you to do something and you have no idea how, so you're searching desperately for an answer to copy so you could do your job and remain ignorant.
Anyway, you need to come up with a way to process your outputs in parallel, and than combine all of the outputs togehter in a single pod.
Fear not, I will show you the way:

Dynamic Fan-in

```
 apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: dynamic-params-fan-in
spec:
  entrypoint: steps
  templates:
######################### steps template ###############################
  - name: steps
    steps:
    - - name: generate
        template: gen-number-list

    # Iterate over the list of numbers generated by the generate step above
    - - name: double
        template: double
        arguments:
          parameters:
          - name: num
            value: "{{item}}"
        withParam: "{{steps.generate.outputs.result}}"

    # Combine all of the last results (num*2)
    - - name: sum
        template: sum
        arguments:
          parameters:
          - name: doubled-numbers
```

UPDF
WWW.UPDF.COM

```
            value: "{{steps.double.outputs.parameters}}"

######################### first template ###############################
    # Generate a list of numbers in JSON format
  - name: gen-number-list
    script:
      image: <registry>/argo/alpine-python:3.11
      command: [python]
      source: |
        import json
        import sys
        import random

        num_of_pods = random.randint(3,10)
        json.dump([random.randint(1,10) for i in range(num_of_pods)], sys.stdout)

######################### second template ###############################
  - name: double
    inputs:
      parameters:
        - name: num
    outputs:
      parameters:
        - name: double-num
          valueFrom:
            path: /mnt/my_volume/result.txt

    volumes:
      - name: my-volume
        emptyDir: {}

    container:
      volumeMounts:
        - name: my-volume
          mountPath: /mnt/my_volume

      image: <registry>/argo/alpine-python:3.11
      command: [sh, -c]
      args: ["echo $(({{inputs.parameters.num}}*2)) >> /mnt/my_volume/result.txt"]

######################### third template ###############################
  - name: sum
    inputs:
      parameters:
      - name: doubled-numbers
    script:
      image: <registry>/argo/alpine-python:3.11
      command: [python]
      source: |
        results = {{ inputs.parameters.doubled-numbers }}
        print(results, type(results))
        print(f"The sum is: {sum([int(obj['double-num']) for obj in results])}")
```

Explainations:

- Lines 20-21: Notice that now the list that the template iterates on is created dynamically.
  The amount of *double* pods that will run is determined by the length of the generated list in the python script, at line 42.
- Lines 28-28: The paramater *steps.<step-name>.outputs.parameters* returns all of the parameters of *<step-name>* as a list of objects (dictionaries).

Example run:

The whole Workflow looks like this in the UI:

UPDF
WWW.UPDF.COM

The **output of the first pod** *(generate)* is*:*

```
[1, 6, 7, 1, 10, 9, 6]
```

Notice that the sum of this array is 40.
The **output of the last pod** is (*sum*) is:

```
[{'double-num': '2'}, {'double-num': '12'}, {'double-num': '14'}, {'double-num': '2'}, {'double-num': '20'}, {'double-num': '18'}, {'double-num': '12'}] <class 'list'>
The sum is: 80
```

Artifacts

Artifacts are simply whole files that you want to save and load with your *Artifact repository.* In our case it's S3-storage. Use Artifacts if you want to **transfer large files** between your Workflow pods.
The *Artifact repository* should be configured beforehand during the installation of Argo. In our case, it means setting our S3 credentials in the Argo *values. yaml.*
In this section, I will assume your Artifact repository is already set up, so you can easily save your files to S3.

Output Artifacts

If your code deals with a lot of data, you might want to transfer big files between pods. Parameters are passed in the k8s YAML to a pod, so it restircts their size in the KB zone.
In order to move around MB's and GB's, you have to save your files to S3 using Artifacts, then load them in the next step.
Let's look at an example:

Artifact outputs

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: artifacts-example-
  name: artifacts-example
spec:
  entrypoint: steps

  templates:
  - name: steps
    steps:
    - - name: gen-artifact
        template: gen-artifact


  - name: gen-artifact
    volumes:
      - name: my-volume
        emptyDir: {}

    outputs:
      artifacts:
```

UPDF
WWW.UPDF.COM

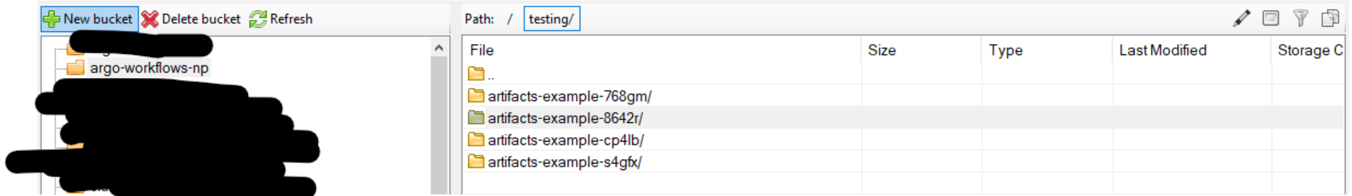
```
          name: big-importent-file
          path: /mnt/my_volume/output.txt
          archive:
            none: { }
          s3:
            key: "testing/{{workflow.name}}/big-importent-file.txt"


    script:
      image: "<registry>/argo/alpine-python:3.11"
volumeMounts:
        - name: my-volume
          mountPath: /mnt/my_volume
      command: [python3]
      source: |
        lines = 500000
        sentence = "blah blah\n"
        with open("/mnt/my_volume/output.txt", "w") as f:
          for i in range(lines):
            f.write(sentence)
```

Explainations:

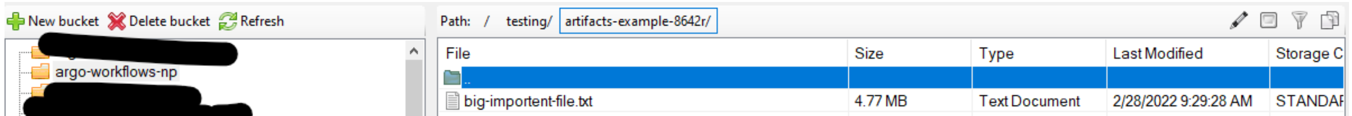
- Lines 18-19: We have to have a volume to use Artifacts.
- Lines 23-24: The name of the artifact and the path of the file you want to save on s3.
- Lines 25-26: Argo **automatically compresses** the Artifacts (tar.gz). I add these lines in order to prevent the compression.
- Lines 27-28: The path (key) in S3 to save the file. It's under the default folder defined beforehand in Argos *values.yaml*.
  I defined the default folder as *argo-workflows-np*, so the full key here is actually: *argo-workflows-np/testing/{{workflow.name}}/big-importent-file.txt*

And in S3:

Under *argo-workflows-np/testing/**



Under *testing/artifacts-example-8642r/**



Input Artifacts

Writing the input is as easy as writing the output, so before I show the full example, let's add a little cool trick:
Let's say you want to create a file on your container in a specefic path. Maybe you want to run a script, but have the script run inside a specific folder (useful for python\nodejs imports).
In order to do that, we can create a file with Artifacts and run it with *command* of the container:

Input Artifacts

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: artifacts-example-
  name: artifacts-example
spec:
  entrypoint: steps

  templates:
######################### steps template ###############################
```

UPDF
WWW.UPDF.COM

```
    name: steps
    steps:
    - - name: gen-artifact
        template: gen-artifact

    - - name: print-artifact
        template: print-artifact
        arguments:
          artifacts:
          - name: results
            from: "{{steps.gen-artifact.outputs.artifacts.big-important-file}}"

######################### first template ###############################
  - name: gen-artifact
    volumes:
      - name: my-volume
        emptyDir: {}

    outputs:
      artifacts:
        - name: big-important-file
          path: /mnt/my_volume/output.txt
          archive:
            none: { }
          s3:
            key: "testing/{{workflow.name}}/big-important-file.txt"

    script:
      image: "<registry>/argo/alpine-python:3.11"
volumeMounts:
        - name: my-volume
          mountPath: /mnt/my_volume
      command: [python3]
      source: |
        lines = 500000
        sentence = "blah blah\n"
        with open("/mnt/my_volume/output.txt", "w") as f:
          for i in range(lines):
            f.write(sentence)
        print("Done")

######################### second template ###############################
  - name: print-artifact
    inputs:
      artifacts:
      - name: results
        path: /home/python_app/results.txt

      - name: start-script
        path: /home/python_app/start_script.py
        raw:
          data: |
            with open("./results.txt", "r") as f:
              print(f.read())
              print("Done")

    container:
      image: "<registry>/argo/alpine-python:3.11"
imagePullPolicy: Always
      command: [python]
      args: ["/home/python_app/start_script.py"]
```

Explainations:

- Lines 19-21: Passing an Artifact. Notice the *from* field instead of *value*.
- Lines 55-57: Injecting the Artifact that was provided to file */home/python_app/results.txt*.
  Notice - we could provide the Artifact in another way. We could use write S3 here under path and load an Artifact straight from S3, the syntax is identical to the output S3 syntax.
  In this case we don't need to pass the Artifact in Steps.
- Lines 59-65: In this way, we insert the text under *raw.data* into the file */home/python_app/start_script.py*. We created a python script next to the *re sults.txt* file!

UPDF
WWW.UPDF.COM

- Lines 70-71: Notice that we don't use *script* template here. We use *container* and run the file we created in *inputs.artifacts[1].*

CronWorkflow and WorkflowTemplate

There are different kinds of Workflows. They are very simillar to a regular Workflow, so you don't have much to learn if you already a master Workflower.

CronWorkflow

The easiest one is CronWorkflow - It's a Workflow that runs according to a Cron.

In the YAML, change *kind*'s value to *CronWorkflow*.
Basically copy-paste your Workflow into the *spec.workflowSpec* value (without *metadata*). Write your Cron in the *spec.schedule* key, and make sure to define the *concurrencyPolicy*.

WorkflowTemplate

Let's say you had written a template and you want to use it in multiple different Workflows.
Instead of copy-pasting it, you can (and should) define it as a WorkflowTemplate and import it to other Workflows!

In the YAML, change *kind*'s value to *WorkflowTemplate* and make sure you have *metadata.name*.
If you want to import a template from this Workflow, just use *templateRef* instead of *template* in *steps.*

Example:

Simple templateRef

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow

metadata:
  generateName: simple-templateref-
  name: simple-templateref

spec:
  entrypoint: main
  templates:
    - name: main
      steps:
        - - name: call-random-numbers-generator
            templateRef:
              name: random-numbers-generator
              template: generator
            arguments:
              parameters:
              - name: min
                value: "10"
              - name: max
                value: "100"
```

Explainations:

- Lines 14-16: This is how we refrence another template.
  This step will use the template named *generator* form the *WorkflowTemplate* named *random-numbers-genertor*.
  It will run the *geneator* template with the provided arguments and parameters.

Final words

That's it, I hope you enjoyed and now you'll be writing **kickass workflows**!

Feel free to add questions here with comments and I would do my best to answer and add corrections.

Written by:

UPDF
WWW.UPDF.COM

נדב פורת -רב"ט