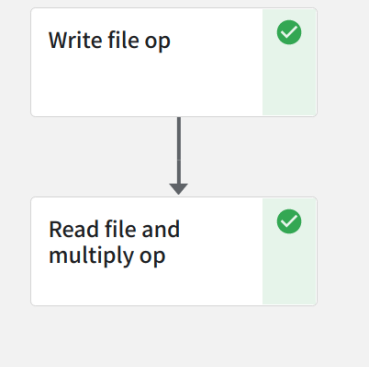
CH 4.5 Kfp 실습 (2)

> 이번 시간에는 kfp 파이프라인을 만들 때, 사용할 수 있는 다양한 기능에 대해서 하나씩 알아보겠습니다.

1. Passing Data between Components by File

> 첫 번째 컴포넌트에서 file 에 data 를 쓴 뒤, 두 번째 컴포넌트에서는 해당 file 로부터 데이터를 읽어 두 수의 곱을 구하는 pipeline 예제입니다.

- Component 간에 데이터를 주고 받는 방법에는 위의 add_example 의 예시처럼 변수를 통해서 넘겨줄 수도 있지만, **데이터의 사이즈가 큰 경우**에는 **파일에 저장한 뒤,** 파일 경로를 전달하는 방식으로 데이터를 넘겨줄 수 있습니다.



▼ data_passing_file.py

```
import kfp
from kfp.components import InputPath, OutputPath, create_component_from_func


# decorator 사용
@create_component_from_func
def write_file_op(
    # _path 라는 suffix 를 붙이고, type annotaion 은 OutputPath 로 선언해야 합니다.
    data_output_path: OutputPath("dict")
):
    # package import 문은 함수 내부에 선언
    import json

    # dict data 선언
    data = {
        "a": 300,
        "b": 10,
    }

    # file write to data_output_path
    with open(data_output_path, "w") as f:
        json.dump(data, f)
```

```
@create_component_from_func
def read_file_and_multiply_op(
    # input 역시, _path 라는 suffix 를 붙이고, type annotation 은 InputPath 로 선언해야 합니다.
    data_input_path: InputPath("dict")
) -> float:
    # package import 문은 함수 내부에 선언
    import json

    # file read to data_output_path
    with open(data_input_path, "r") as f:
        data = json.load(f)

    # multiply
    result = data["a"] * data["b"]

    print(f"Result: {result}")

    return result


@kfp.dsl.pipeline(name="Data Passing by File Example")
def data_passing_file_pipeline():
    write_file_task = write_file_op()
    _ = read_file_and_multiply_op(write_file_task.outputs["data_output"])


if __name__ == "__main__":
    kfp.compiler.Compiler().compile(
        data_passing_file_pipeline,
        "./data_passing_file_pipeline.yaml"
    )
```

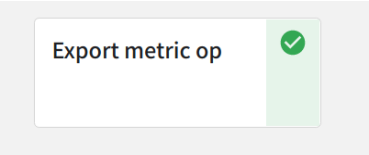
- **정해진 Rule 대로** component, pipeline 을 작성하면 kubeflow pipeline SDK 가 관련 부분을 file passing 으로 인식하여 pipeline 컴파일 시에 관련 처리를 자동화해줍니다.
  - Rule :
    - component 의 argument 를 선언할 때, argument name 의 suffix 로 `_path` 를 붙이고, type annotation 을 `kfp.components.InputPath` 혹은 `kfp.components.OutputPath` 로 선언합니다.
- 컴파일된 Workflow yaml 파일을 확인하여, `add_pipeline.yaml` 와 어떤 점이 다른지 확인해봅니다.
  - yaml 파일의 `dag` 부분을 보면, `arguments` 의 value 로 `parameter` 가 아닌, `artifacts` 로 선언되어 있는 것을 확인할 수 있습니다.
- UI 를 통해 pipeline 업로드 후, 실행해봅니다. 컴포넌트 간의 input, output 이 어떻게 연결되는지 확인합니다.

---

2. Export Metrics in Components

> 컴포넌트에서 metrics 를 남기는 pipeline 예제입니다.

- 하나의 컴포넌트에서 metrics 를 export 하는 예제입니다.



▼ export_metrics.py

```
import kfp
from kfp.components import OutputPath, create_component_from_func


@create_component_from_func
def export_metric_op(
    mlpipeline_metrics_path: OutputPath("Metrics"),
):
    # package import 문은 함수 내부에 선언
    import json

    # 아래와 같이 정해진 형태로, key = "metrics", value = List of dict
    # 단, 각각의 dict 는 "name", "numberValue" 라는 key 를 가지고 있어야 합니다.
    # "name" 의 value 로 적은 string 이 ui 에서 metric 의 name 으로 parsing 됩니다.
    # 예시이므로, 특정 모델에 대한 값을 직접 계산하지 않고 const 로 작성하겠습니다.
    metrics = {
        "metrics": [
            # 개수는 따로 제한이 없습니다. 하나의 metric 만 출력하고 싶다면, 하나의 dict 만 원소로 갖는 list 로 작성해주시면 됩니다.
            {
                "name": "auroc",
                "numberValue": 0.8,  # 당연하게도 scala value 를 할당받은 python 변수를 작성해도 됩니다.
            },
            {
                "name": "f1",
                "numberValue": 0.9,
                "format": "PERCENTAGE",
                # metrics 출력 시 포맷을 지정할 수도 있습니다. Default 는 "RAW" 이며 PERCENTAGE 를 사용할 수도 있습니다.
            },
        ],
    }

    # 위의 dict 타입의 변수 metrics 를 mlpipeline_metrics_path 에 json.dump 합니다.
    with open(mlpipeline_metrics_path, "w") as f:
        json.dump(metrics, f)


@kfp.dsl.pipeline(name="Export Metrics Example")
def export_metrics_pipeline():
    write_file_task = export_metric_op()


if __name__ == "__main__":
    kfp.compiler.Compiler().compile(
        export_metrics_pipeline,
        "./export_metrics_pipeline.yaml"
    )
```

- **정해진 Rule 대로** component, pipeline 을 작성하면 kubeflow pipeline SDK 가 관련 부분을 export metrics 으로 인식하여 pipeline 컴파일 시에 관련 처리를 자동화해줍니다.
    - Rule :
        - component 의 argument 를 선언할 때, argument name 은 `mlpipeline_metrics_path` 여야 하며, type annotation 은 `OutputPath('Metrics')` 로 선언합니다.

        - component 내부에서 metrics 를 위의 코드의 주석의 룰을 지켜 선언하고 json dump 하여 사용합니다.
- 컴파일된 Workflow yaml 파일을 확인하여, `add_pipeline.yaml` 와 어떤 점이 다른지 확인해봅니다.
    - yaml 파일의 `dag` 부분을 보면, `arguments` 의 value 로 `parameter` 가 아닌, `artifacts` 로 선언되어 있는 것을 확인할 수 있습니다.
- 주의사항
    - metrics 의 name 은 반드시 다음과 같은 regex pattern 을 만족해야 합니다.
        - `^[a-zA-Z]([-_a-zA-Z0-9]{0,62}[a-zA-Z0-9])?$`
    - metrics 의 numberValue 의 value 는 반드시 **numeric type** 이어야 합니다.
- UI 를 통해 pipeline 업로드 후, 실행해봅니다. Run output 을 눌러 Metrics 가 어떻게 출력되는지 확인합니다.
    - metrics 를 출력한 component 의 이름, 그리고 해당 component 에서의 key, value 를 확인할 수 있습니다.

---

3. Use resources in Components

> 하나의 컴포넌트에서, k8s resource 들을 직접 지정하여 사용하는 방법을 간단히 다룹니다.
> 컴포넌트 별로, 필요한 리소스를 할당할 수 있습니다.
> 이번 파트는 실습은 하지 않고 사용법만 알아보겠습니다.

- CPU, Memory 할당
    - 다음과 같은 형태로 `ContainerOp` 에 메소드 체이닝 형태로 작성하면, 해당 component 를 run 할 때, pod 의 리소스가 지정되어 생성

```
@dsl.pipeline()
def pipeline():
...
  training_task = training_op(learning_rate, num_layers, optimizer).set_cpu_request(2).set_cpu_limit(4).s
et_memory_request('1G').set_memory_limit('2G')
...
```

- GPU 할당
    - cpu, memory 와 동일한 방법으로 작성

```
@dsl.pipeline()
def pipeline():
...
  training_task = training_op(learning_rate, num_layers, optimizer).set_gpu_limit(1)
...
```

    - 단, 해당 component 의 base_image 에 cuda, cudnn, tensorflow-gpu 등 GPU 를 사용할 수 있는 docker image 를 사용해야 정상적으로 사용 가능
- PVC 할당
    - k8s 의 동일한 namespace 에 pvc 를 미리 생성해둔 뒤, 해당 pvc 의 name 을 지정하여 다음과 같은 형태로 `ContainerOp` 의 argument 로 직접 작성

```
@dsl.pipeline()
def pipeline():
...
```

```
  vop = dsl.VolumeOp(
      name="v1",
      resource_name="mypvc",
      size="1Gi"
  )

  use_volume_op = dsl.ContainerOp(
      name="test",
      ...
      pvolumes={"/mnt": vop.volume} # 이렇게 ContainerOp 생성 시, argument 로 지정
  )
...
```

- 혹은, 다음과 같이 `add_pvolumes` 를 사용하여 작성

```
@dsl.pipeline()
def pipeline():
...
  vop = dsl.VolumeOp(
      name="v2",
      resource_name="mypvc",
      size="1Gi"
  )

  use_volume_op = dsl.ContainerOp(
      name="test"
  )

  # 이렇게 위의 CPU, MEMORY, GPU 처럼 내장 메소드 사용
  use_volume_task = use_volume_op("name").add_pvolumes({"/mnt": vop.volume})
```

- Secret 를 Env variable 로 사용
  - k8s 의 동일한 namespace 에 secret 를 미리 생성해둔 뒤, 해당 secret 의 name 과 value 지정하여 다음과 같은 형태로 `add_env_variable` 사용하여 작성

```
@dsl.pipeline()
def pipeline():
...
  env_var = V1EnvVar(name='example_env', value='env_variable')

  use_secret_op = dsl.ContainerOp(
      name="test"
  )

  use_secret_task = use_secret_op("name").add_env_variable(env_var)
...
```

  - 사용할 정보를 담은 secret 을 미리 만들어두고, 위의 예시처럼 `add_env_variable` 함수를 사용해서 component(pod) 에서 붙이면, component python code 내부에서는 그냥 `os.environ.get()` 등을 사용하여 활용할 수 있습니다.

- kfp Pipeline, component 에서 자주 활용하는 쿠버네티스의 리소스별 사용법은 위와 같으나, 이외에도 대부분의 k8s resource 를 사용할 수 있습니다.
  - 더 다양한 기능은 **Official API Reference** 를 참고해주시기 바랍니다.

> kfp.dsl._container_op - Kubeflow Pipelines documentation
> """Takes the `ContainerOp` class and proxy the PendingDeprecation properties in `ContainerOp` to the `Container` instance. """
> https://kubeflow-pipelines.readthedocs.io/en/stable/_modules/kfp/dsl/_container_op.html