Creating classic templatesbookmark

cloud.google.com/dataflow/docs/guides/templates/creating-templates

Dataflow templates use *runtime parameters* to accept values that are only available during pipeline execution. To customize the execution of a templated pipeline, you can pass these parameters to functions that run within the pipeline (such as a `DoFn`).

To create a template from your Apache Beam pipeline, you must modify your pipeline code to support runtime parameters:

- Use `ValueProvider` for all pipeline options in classic templates that you want to set or use at runtime.
- Call I/O methods that accept runtime parameters wherever you want to parameterize your pipeline.
- Use `DoFn` objects that accept runtime parameters.

Then, create and stage your template.

Runtime parameters and the ValueProvider interface

The `ValueProvider` interface allows pipelines to accept runtime parameters. Apache Beam provides three types of `ValueProvider` objects.

Name	Description
`RuntimeValueProvider`	`RuntimeValueProvider` is the default `ValueProvider` type. `RuntimeValueProvider` allows your pipeline to accept a value that is only available during pipeline execution. The value is not available during pipeline construction, so you can't use the value to change your pipeline's workflow graph. You can use `isAccessible()` to check if the value of a `ValueProvider` is available. If you call `get()` before pipeline execution, Apache Beam returns an error: `Value only available at runtime, but accessed from a non-runtime context.` Use `RuntimeValueProvider` when you do not know the value ahead of time.

`StaticValueProvider`	`StaticValueProvider` allows you to provide a static value to your pipeline. The value is available during pipeline construction, so you can use the value to change your pipeline's workflow graph. Use `StaticValueProvider` when you know the value ahead of time. See the StaticValueProvider section for examples.
`NestedValueProvider`	`NestedValueProvider` allows you to compute a value from another `ValueProvider` object. `NestedValueProvider` wraps a `ValueProvider`, and the type of the wrapped `ValueProvider` determines whether the value is accessible during pipeline construction. Use `NestedValueProvider` when you want to use the value to compute another value at runtime. See the NestedValueProvider section for examples.

The Dataflow runner does not support `ValueProvider` options for Pub/Sub topics and subscription parameters. If you require Pub/Sub options in your runtime parameters, switch to using Flex Templates.

Modifying your code to use runtime parameters

This section walks through how to use `ValueProvider`, `StaticValueProvider`, and `NestedValueProvider`.

Using ValueProvider in your pipeline options

Use `ValueProvider` for all pipeline options that you want to set or use at runtime.

For example, the following `WordCount` code snippet does not support runtime parameters. The code adds an input file option, creates a pipeline, and reads lines from the input file:

```
 class WordcountOptions(PipelineOptions):
  @classmethod
  def _add_argparse_args(cls, parser):
   parser.add_argument(
     '--input',
     default='gs://dataflow-samples/shakespeare/kinglear.txt',
     help='Path of the file to read from')
   parser.add_argument(
     '--output',
     required=True,
     help='Output file to write results to.')
 pipeline_options = PipelineOptions(['--output', 'some/output_path'])
 p = beam.Pipeline(options=pipeline_options)  wordcount_options =
pipeline_options.view_as(WordcountOptions)
 lines = p | 'read' >> ReadFromText(wordcount_options.input)
```

To add runtime parameter support, modify the input file option to use `ValueProvider`.

Replace `add_argument` with `add_value_provider_argument`.

```
class WordcountOptions(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
      # Use add_value_provider_argument for arguments to be templatable
      # Use add_argument as usual for non-templatable arguments
      parser.add_value_provider_argument(
          '--input',
          default='gs://dataflow-samples/shakespeare/kinglear.txt',
          help='Path of the file to read from')
      parser.add_argument(
          '--output',
          required=True,
          help='Output file to write results to.')
  pipeline_options = PipelineOptions(['--output', 'some/output_path'])
  p = beam.Pipeline(options=pipeline_options)  wordcount_options =
pipeline_options.view_as(WordcountOptions)
  lines = p | 'read' >> ReadFromText(wordcount_options.input)
```

Using ValueProvider in your functions

To use runtime parameter values in your own functions, update the functions to use `ValueProvider` parameters.

The following example contains an integer `ValueProvider` option, and a simple function that adds an integer. The function depends on the `ValueProvider` integer. During execution, the pipeline applies `MySumFn` to every integer in a `PCollection` that contains `[1, 2, 3]`. If the runtime value is 10, the resulting `PCollection` contains `[11, 12, 13]`.

```
  import apache_beam as beam
  from apache_beam.options.pipeline_options import PipelineOptions
  from apache_beam.options.value_provider import StaticValueProvider
  from apache_beam.io import WriteToText
  class UserOptions(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
      parser.add_value_provider_argument('--templated_int', type=int)
  class MySumFn(beam.DoFn):
    def __init__(self, templated_int):
      self.templated_int =

 templated_int


    def process(self, an_int):
      yield self.templated_int.get() +

 an_int

  pipeline_options = PipelineOptions()
  p = beam.Pipeline(options=pipeline_options)  user_options =
pipeline_options.view_as(UserOptions)
  sum = (p
         | 'ReadCollection' >> beam.io.ReadFromText(
           'gs://some/integer_collection')
         | 'StringToInt' >> beam.Map(lambda w: int(w))
         | 'AddGivenInt' >> beam.ParDo(MySumFn(user_options.templated_int))
         | 'WriteResultingCollection' >> WriteToText('some/output_path'))
```

Using StaticValueProvider

To provide a static value to your pipeline, use `StaticValueProvider` .

This example uses `MySumFn` , which is a `DoFn` that takes a `ValueProvider<Integer>` . If you know the value of the parameter ahead of time, you can use `StaticValueProvider` to specify your static value as a `ValueProvider` .

This code gets the value at pipeline runtime:

```
  beam.ParDo(MySumFn(user_options.templated_int))
```

Instead, you can use `StaticValueProvider` with a static value:

```
  beam.ParDo(MySumFn(StaticValueProvider(int,10)))
```

You can also use `StaticValueProvider` when you implement an I/O module that supports both regular parameters and runtime parameters. `StaticValueProvider` reduces the code duplication from implementing two similar methods.

In this example, there is a single constructor that accepts both a `string` or a `ValueProvider` argument. If the argument is a `string` , it is converted to a `StaticValueProvider` .

```
class Read():
  def __init__(self, filepattern):
    if isinstance(filepattern, basestring):
      # Create a StaticValueProvider from a regular string parameter
      filepattern = StaticValueProvider(str, filepattern)
    self.filepattern = filepattern
```

Using NestedValueProvider

To compute a value from another `ValueProvider` object, use `NestedValueProvider`.

`NestedValueProvider` takes a `ValueProvider` and a `SerializableFunction` translator as input. When you call `.get()` on a `NestedValueProvider`, the translator creates a new value based on the `ValueProvider` value. This translation allows you to use a `ValueProvider` value to create the final value that you want.

Note: `NestedValueProvider` accepts only one value input. You can't use a `NestedValueProvider` to combine two different values.
In the following example, the user provides the filename `file.txt`. The transform prepends the path `gs://directory_name/` to the filename. Calling `.get()` returns `gs://directory_name/file.txt`.

```
  public interface WriteIntsOptions extends PipelineOptions {
      // New runtime parameter, specified by the --fileName
      // option at runtime.
      ValueProvider<String> getFileName();
      void setFileName(ValueProvider<String> value);
  }
  public static void main(String[] args) {
     WriteIntsOptions options =
          PipelineOptionsFactory.fromArgs(args).withValidation()
            .as(WriteIntsOptions.class);
    Pipeline p = Pipeline.create(options);    p.apply(Create.of(1, 2, 3))
     // Write to the computed complete file path.
     .apply("OutputNums", TextIO.write().to(NestedValueProvider.of(
        options.getFileName(),
        new SerializableFunction<String, String>() {
          @Override
          public String apply(String file) {
            return "gs://directoryname/" + file;
          }
        })));    p.run();
  }
```

Metadata

You can extend your templates with additional metadata so that custom parameters are validated when the template executes. If you want to create metadata for your template, you need to:

1. Create a JSON-formatted file named `<template-name>_metadata` using the parameters from the table below.
   Note: Do not name the file you create `<template-name>_metadata.json`. While the file contains JSON, it cannot end in the `.json` file extension.
2. Store the JSON-formatted file in Cloud Storage in the same folder as the template.
   Note: Store the template as `<template-name>` and store its metadata file as `<template-name>_metadata`.

Metadata parameters

Parameter Key		Required	Description of the value
`name`		Yes	The name of your template.
`description`		No	A short paragraph of text describing the templates.
`parameters`		No. Defaults to an empty array.	An array of additional parameters that will be used by the template.
	`name`	Yes	The name of the parameter used in your template.
	`label`	Yes	A human readable label that will be used in the UI to label the parameter.
	`helpText`	Yes	A short paragraph of text describing the parameter.
	`isOptional`	No. Defaults to false.	`false` if the parameter is required and `true` if the parameter is optional. If you do not include this parameter key for your metadata, the metadata becomes a required parameter.
	`regexes`	No. Defaults to an empty array.	An array of POSIX-egrep regular expressions in string form that will be used to validate the value of the parameter. For example: `["^[a-zA-Z][a-zA-Z0-9]+"]` is a single regular expression that validates that the value starts with a letter and then has one or more characters.

Example metadata file

JavaPython

The Dataflow service uses the following metadata to validate the wordcount template's custom parameters:

```
{
  "description": "An example pipeline that counts words in the input file.",
  "name": "Word Count",
  "parameters": [
    {
      "regexes": [
        "^gs:\\/\\/[^\\n\\r]+$"
      ],
      "name": "input",
      "helpText": "Path of the file pattern glob to read from. ex: gs://dataflow-
samples/shakespeare/kinglear.txt",
      "label": "Input Cloud Storage file(s)"
    },
    {
      "regexes": [
        "^gs:\\/\\/[^\\n\\r]+$"
      ],
      "name": "output",
      "helpText": "Path and filename prefix for writing output files. ex:
gs://MyBucket/counts",
      "label": "Output Cloud Storage file(s)"
    }
  ]
}
```

You can download metadata files for the Google-provided templates from the Dataflow template directory.

Pipeline I/O and runtime parameters

Some I/O connectors contain methods that accept `ValueProvider` objects. To determine support for I/O connectors and their methods, see the API reference documentation for the connector. The following I/O connectors accept runtime parameters:

File-based IOs: `textio`, `avroio`, `tfrecordio`

Creating and staging templates

After you write your pipeline, you must create and stage your template file.

Note: After you create and stage a template, the staging location contains additional files that are necessary to run your template. If you delete the staging location, the template fails to run.
See the following examples on how to stage a template file:

This Python command creates and stages a template at the Cloud Storage location specified with `--template_location`.

Note: If you use the Apache Beam SDK for Python 2.15.0 or later, you must also specify `--region`.

```
python -m examples.mymodule \
  --runner DataflowRunner \
  --project PROJECT_ID \
  --staging_location gs://BUCKET_NAME/staging \
  --temp_location gs://BUCKET_NAME/temp \
  --template_location gs://BUCKET_NAME/templates/TEMPLATE_NAME \
  --region REGION
```

Verify that the `template_location` path is correct. Replace the following:

- `examples.mymodule` : your Python module
- `PROJECT_ID` : your project ID
- `BUCKET_NAME` : the name of your Cloud Storage bucket
- `TEMPLATE_NAME` : the name of your template
- `REGION` : the regional endpoint to deploy your Dataflow job

After you create and stage your template, your next step is to execute the template.

Rate and review