Common Workflow Language

Introduction

Common Workflow Language (CWL) is a specification that allows you to connect together command line tools together in workflows.

Setup

The following installs the reference implementation of CWL onto a base Ubuntu machine...

```
$ sudo apt install build-essential virtualenv python-dev
$ virtualenv -p python2 venv
$ pip install cwlref-runner
$ cwl-runner --version
/home/user/venv/bin/cwl-runner 1.0.20181201184214
```

Basics

Begin by defining the command line tools to run…

```
# cat.cwl
cwlVersion: v1.0
class: CommandLineTool
baseCommand: cat
inputs:
  file1:
    type: File
    inputBinding:
      position: 1
  file2:
    type: File
    inputBinding:
      position: 2
outputs:
  combined_out:
    type: stdout
```

```
# linecount.cwl
cwlVersion: v1.0
class: CommandLineTool
baseCommand: wc
arguments: [-l]
inputs:
  file:
    type: File
    inputBinding:
      position: 1
outputs:
  line_count:
    type: stdout
```

Then, create a workflow that uses these tools…

```
# workflow.cwl
cwlVersion: v1.0
class: Workflow
inputs:
  file1: File
  file2: File
outputs:
  lines:
    type: File
    outputSource: counter/line_count
steps:
  combine:
    run: cat.cwl
    in:
      file1: file1
      file2: file2
    out: [combined_out]
  counter:
    run: linecount.cwl
    in:
      file: combine/combined_out
    out: [line_count]
```

Then, create the file that has the inputs for your workflow…

```
# workflow_inputs.yaml
file1:
  class: File
  path: /proc/version
file2:
  class: File
  path: /proc/stat
```

Finally, run the actual workflow by using cwl-runner…

```
$ cwl-runner workflow.cwl workflow_inputs.yml
/home/user/venv/bin/cwl-runner 1.0.20181201184214
Resolved 'workflow.cwl' to 'file:///home/user/workflow.cwl'
[workflow ] start
[workflow ] starting step combine
[step combine] start
[job combine] /tmp/tmp64v2ef$ cat \
    /tmp/tmpYS_elo/stg7a0b9e50-b05f-4eda-8e8d-7714131f33f6/version \
    /tmp/tmpYS_elo/stgbce589ee-9791-48a7-ad6a-5ca9c1d4baeb/stat >
/tmp/tmp64v2ef/0df11f3aeb794a61d2156191b92f82824df8dbc8
[job combine] completed success
[step combine] completed success
```

```
[workflow ] starting step counter
[step counter] start
[job counter] /tmp/tmpwceXJH$ wc \
    -l \

/tmp/tmpxcs1rL/stg659a705b-3eee-4b72-a8ab-f7132688816b/0df11f3aeb794a61d215
6191b92f82824df8dbc8 >
/tmp/tmpwceXJH/905f60d29a83a3c086fd1d01a53ff192f54e8e19
[job counter] completed success
[step counter] completed success
[workflow ] completed success
{
    "lines": {
        "checksum": "sha1$42680223ea6644015f798ec9b29c3d10462cbaf7",
        "basename": "905f60d29a83a3c086fd1d01a53ff192f54e8e19",
        "location":
"file:///home/user/905f60d29a83a3c086fd1d01a53ff192f54e8e19",
        "path": "/home/user/905f60d29a83a3c086fd1d01a53ff192f54e8e19",
        "class": "File",
        "size": 99
    }
}
Final process status is success
```

The output of the workflow is spit out by cwl-runner’s output. In this case, it’s a path to a file...

```
$ cat /home/user/905f60d29a83a3c086fd1d01a53ff192f54e8e19
10
/tmp/tmpxcs1rL/stg659a705b-3eee-4b72-a8ab-f7132688816b/0df11f3aeb794a61d215
6191b92f82824df8dbc8
```

The general form of running a CWL file is…

```
cwl-runner [cwl_file] [inputs_for_cwl_file]
```

It works the same for both command line tool scripts and workflow scripts. Specify the script and the input file and it should spit out a result.

Command Line Script

Before being able to build up a dependency of tasks together, you need to define what the tasks are along with what inputs they expect and what outputs they produce. CWL treats tasks as command-line tools, and their definitions are written in YAML (or JSON) syntax.

For example, imagine you wanted to create a task that takes 2 input files and concatenates them together...

```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0       # Version of CWL spec
class: CommandLineTool # Script class of commandline tool
baseCommand: cat       # Command to run
inputs:                # List of inputs into command
  file1:              # Input1 ID
    type: File        # Input1 datatype
    inputBinding:       # How input should appear to command
      position: 1       #  (indicates that it should appear first)
  file2:              # Input2 ID
    type: File        # Input2 datatype
    inputBinding:       # How input should appear to command
      position: 2       #  (indicates that it should appear second)
outputs:              # List of outputs from command
  combined_out:         # Output1 ID
    type: stdout        # Output1 type (where it came from)
```

> **NOTE**: baseCommand is a single command, but it can also be an array. If it’s an array, the first item is the command and subsequent items are hardcoded args (passed in before any of the inputs). Alternatively, you can use the arguments tag to hardcode arrays.

You can test the above script by placing it in a file (cat.cwl), building another file with the inputs required to run it (cat_inputs.yml), and then test it by invoking cwl-runner...

```
file1:
  class: File
  path: /proc/version
file2:
  class: File
  path: /proc/stat
```

```
$ cwl-runner cat.cwl cat_inputs.yml
/home/user/venv/bin/cwl-runner 1.0.20181201184214
Resolved 'cat.cwl' to 'file:///home/user/cat.cwl'
[job cat.cwl] /tmp/tmpm4YP9U$ cat \
    /tmp/tmpI9xB7Y/stg14d184bc-01ad-4377-8cc7-16a9164296b2/version \
    /tmp/tmpI9xB7Y/stgc972e2a5-7395-4569-99a7-b828db1bbe0c/stat >
```

```
/tmp/tmpm4YP9U/0df11f3aeb794a61d2156191b92f82824df8dbc8
[job cat.cwl] completed success
{
    "combined_out": {
        "checksum": "sha1$b0f4ef4b1acacdde701fd1fadb854507504e2374",
        "basename": "0df11f3aeb794a61d2156191b92f82824df8dbc8",
        "location":
"file:///home/user/0df11f3aeb794a61d2156191b92f82824df8dbc8",
        "path": "/home/user/0df11f3aeb794a61d2156191b92f82824df8dbc8",
        "class": "File",
        "size": 927
    }
}
Final process status is success
```

Base Command

The command to run is specified as baseCommand...

```
cwlVersion: v1.0       # Version of CWL spec
class: CommandLineTool # Workflow type/class
baseCommand: cat       # Command to run
inputs:                # List of inputs into command
  ...
outputs:               # List of outputs from command
  ...
```

If baseCommand is an array rather than a string, the first item is the command and subsequent items are hardcoded args (passed in before any of the inputs)...

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: [cat, -b]
inputs:
  ...
outputs:
  ...
```

Alternatively, you can use the arguments tag for hardcode arguments…

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: cat
```

```
arguments: [-b]
inputs:
  ...
outputs:
  ...
```

Inputs

The inputs of a tool control how the tool gets run. An input is defined by an ID, type, and optional binding information. For example, the following could be used as inputs for a Linux “tail” command…

```
inputs:                 # List of inputs for tail command
  follow_flag:          # Input1 ID (used for following output)
    type: boolean       # Input1 type
    inputBinding:       # How Input1 should appear to command
      prefix: -f         #  (what shows if input set to true)
      position: 1        #  (indicates that it should appear as arg1)
  initial_lines:        # Input2 ID (used for initial line count)
    type: int           # Input2 datatype
    inputBinding:       # How Input2 should appear to command
      prefix: -lines=    #  (what shows up before the integer input)
      separate: false    #  (keep the prefix and the arg together)
      position: 2        #  (indicates that it should appear as arg2)
```

The input file for the following set of inputs may look like this…

```
follow_flag: true
initial_lines: 10
```

Input Types

The following input types are allowed:

- string (primitive)
- int (primitive)
- long (primitive)
- float (primitive)
- double (primitive)
- boolean (???)
  NOTE: boolean has a special meaning in inputBinding.
- null (primitive)
- array (complex)
- record (complex)
- File (special)

- Directory (special)
- Any (special)

“Special” input types require that you specify a “class” for that in your inputs YAML. For example, if you have a tool that takes in a file input…

```
inputs:
  csv_file:
    type: File
    inputBinding:
      prefix: --file=
      separate: false
      position: 1
```

You’d specify it in your input file as such…

```
csv_file:
  class: File
  path: my_file.csv
```

Input Binding

An input binding defines where and how an input appears on the command-line tool. The input binding is specified using the inputBinding tag, and there are 3 parameters that it can have: prefix, separate, and position. For example…

```
inputs:
  initial_lines:
    type: int
    inputBinding:
      prefix: -lines=
      separate: false
      position: 2
```

- prefix → text to include before the argument (defaults to nothing)
- separate → if the prefix and the argument should be separate args (defaults to true)
- position → where the input be in the command line

  NOTE: For boolean types, prefix has a special meaning. If the boolean input is…
  - true, the prefix is added as the argument to the tool
  - false, nothing is added.

  NOTE: The positions of inputs are not absolute. They’re relative to each other and you don’t even have to specify them sequentially. Inputs with position 1, 2, 3 will appear the same way as inputs with positions 1,7,9.

If an input binding isn't included, that input won't be passed to the tool…

```
inputs:
  follow_flag:
    type: boolean
    # Notice how there is no inputBinding for Input1. That means that
    # this input won't get passed to the tool.
  initial_lines:
    type: int
    inputBinding:
      prefix: -lines=
      separate: false
      position: 2
```

Array Inputs

Inputs can be arrays…

```
inputs:
  filesA:
    type: string[]
    inputBinding:
      prefix: -A
      position: 1
  filesB:
    type:
      type: array
      items: string
      inputBinding:
        prefix: -B=
        separate: false
    inputBinding:
      position: 2
  filesC:
    type: string[]
    inputBinding:
      prefix: -C=
      itemSeparator: ","
      separate: false
      position: 4
```

The argument that gets fed into the tool changes depending on where inputBinding was placed. For example, imagine the array [x, y, z] was passed in to each input above.

filesA and filesC would get interpreted have the a single prefix followed by the array of items, but filesB would have the prefix applied to each item…

- filesA → -A x y z
- filesB → -B=x -B=y -B=z
- filesC → -C=x,y,z

  NOTE: This example was lifted directly from https://www.commonwl.org/user_guide/09-array-inputs/index.html

Optional Inputs

If you want to allow the input to be specified but ignore it if it wasn't specified (optional input), suffix the input type with a question mark…

```
inputs:
  csv_input:
    type: File?  # Notice the question mark. If there is no entry for
                 # csv_input in the input file, this input gets ignored.
    inputBinding:
      prefix: --file=
      separate: false
      position: 1
```

Ignore Inputs

If you want to ignore the input completely when you run the tool, don't include the inputBinding tag…

```
inputs:
  csv_input:
    type: File
    # Notice that inputBinding does not exist for this input.
```

Outputs

The outputs of a tool are either output files, extracted contents of output files, stdout, and/or stderr. For example, the following could be used as outputs from a Linux "cat" command…

```
outputs:                # List of outputs from command
  combined_out:           # Output1 ID
    type: stdout          # Output1 type (where it came from)
```

Output Types

In most cases your output types will be:

- File

- stdout
- stderr

NOTE: It looks like stdout/stderr aren't listed under the CWL specification. They may be special values?

File Output

File outputs require that you target the specific file that gets generated using the outputBinding tag…

```
outputs:
  extracted_file:
    type: File
    outputBinding:
      glob: output_*.csv # put in a exact name or a glob pattern
```

Additionally, you can set the output filename to an input parameter using parameter references…

```
inputs:
 inputfile:
    type: File
    inputBinding:
      prefix: --file
      position: 1
outputs:
  outputfile:
    type: File
    outputBinding:
      glob: $(inputs.inputfile.path) # reference path of inputfile input
```

NOTE: Parameter references allow you to access parameters at runtime. They aren't limited to what you define in your script/files (e.g. you can use runtime.outdir to get the directory of where your outputs should be going. For more information on parameter references, see https://www.commonwl.org/user_guide/06-params/index.html and https://www.commonwl.org/user_guide/08-arguments/index.html.

Multiple File Outputs

If the tool generates multiple file outputs (e.g. split archive), you can type the output as an array…

```
outputs:
  output:
    type:
      type: array
      items: File
    outputBinding:
      glob: "output.part*.rar"
```

The above example will collect all split archive files as a single array output (e.g. [output.part01.rar, output.part02.rar, output.part03.rar])

Standard out/error Output

stdout/stderr can be grabbed as an output as well…

```
outputs:                # List of outputs from command
  combined_out:           # Output1 ID
    type: stdout          # Output1 type (where it came from)
```

Advanced Features

Composite/union input types: https://www.commonwl.org/user_guide/11-records/index.html
Environment variables: https://www.commonwl.org/user_guide/12-env/index.html
Manipulate via expressions: https://www.commonwl.org/user_guide/13-expressions/index.html

Workflow Script

A workflow allows you to build a dependency of tasks (either command-line tools or other workflows). Similar to how command line tools are defined, you'll need to specify inputs and outputs for the workflow. The difference is that outputs of the workflow are linked to the outputs of one of the tasks it runs.

Similarly, the inputs to any of the tasks in the workflow can be references to

- inputs to the workflow.
- outputs of other tasks in the workflow.

That's essentially how the dependency chain is defined. Tasks with inputs coming from the workflow directly are run first, then tasks that depend on the outputs of those tasks, and so on and so forth.

For example...

```
#!/usr/bin/env cwl-runner
```

```
cwlVersion: v1.0       # Version of CWL spec
class: Workflow        # Script class of Workflow

# Workflow inputs
inputs:                # List of inputs into workflow
  tarball: File                   # Input1 ID to type
  file_to_extract: string         # Input2 ID to type

# Workflow outputs
outputs:               # List of outputs from workflow
  compiled_class:                 # Output1 ID
    type: File                      # Output1 type
    outputSource: compile/classfile # Output1 source (from compile task)

# Steps that workflow runs through
steps:
  # The first task in the workflow is to untar. Notice that the inputs
  # for this task are being taken from the inputs to the workflow. That
  # means that this task runs first (it has no dependencies).
  untar:                                      # Task1 ID
    run: tar-param.cwl                        # Task1 script
    in:                                       # Task1 inputs
      tarfile: tarball                         # Task1 tarfile input
      extractfile: file_to_extract             # Task1 extractfile input
    out: [extracted_file]                     # Task1 expected outputs

  # The second task in the workflow is to compile. Notice that the input
  # for this task is the output of the task above (untar/extract). That
  # means that this task is dependent on the task above -- it has to wait
  # for it to finish before it can start.
  #
  # Also notice that the output from this task is the output for this
  # workflow (compile/classfile).
  compile:                                    # Task2 ID
    run: arguments.cwl                        # Task2 script
    in:                                       # Task2 inputs
      src: untar/extracted_file                # Task2 src input
    out: [classfile]                          # Task2 expected outputs
```

NOTE: The example above is from

https://www.commonwl.org/user_guide/21-1st-workflow/index.html, but it’s been commented to highlight the important pieces.