Common Workflow Language, v1.0

DOI:
10.6084/m9.figshare.3115156.v2

Document Version
Final published version

A Gentle Introduction to the Common Workflow Language

Hello!

This guide will introduce you to writing tool wrappers and workflows using the Common Workflow Language (CWL). This guide describes the current stable specification, version 1.0.

Note: This document is a work in progress. Not all features are covered, yet.

Table of contents

1. Introduction

CWL is a way to describe command line tools and connect them together to create workflows. Because CWL is a specification and not a specific piece of software, tools and workflows described using CWL are portable across a variety of platforms that support the CWL standard.

CWL has roots in "make" and many similar tools that determine order of execution based on dependencies between tasks. However unlike "make", CWL tasks are isolated and you must be explicit about your inputs and outputs. The benefit of explicitness and isolation are flexibility, portability, and scalability: tools and workflows described with CWL can transparently leverage technologies such as Docker, be used with CWL implementations from different vendors, and is well suited for describing large-scale workflows in cluster, cloud and high performance computing environments where tasks are scheduled in parallel across many nodes.

2. Wrapping Command Line Tools

2.1 First example

The simplest "hello world" program. This accepts one input parameter, writes a message to the terminal or job log, and produces no permanent output. CWL documents are written in JSON (http://json.org) or YAML (http://yaml.org), or a mix of the two.

1st-tool.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: echo
inputs:
  message:
    type: string
    inputBinding:
      position: 1
outputs: []
```

Use a YAML object in a separate file to describe the input of a run:

echo-job.yml

```
message: Hello world!
```

Now invoke `cwl-runner` with the tool wrapper and the input object on the command line:

```
$ cwl-runner 1st-tool.cwl echo-job.yml
[job 140199012414352] $ echo 'Hello world!'
Hello world!
Final process status is success
```

What's going on here? Let's break it down:

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

The `cwlVersion` field indicates the version of the CWL spec used by the document. The `class` field indicates this document describes a command line tool.

```
baseCommand: echo
```

The `baseCommand` provides the name of program that will actually run (echo)

```
inputs:
  message:
    type: string
    inputBinding:
      position: 1
```

The `inputs` section describes the inputs of the tool. This is a list of input parameters and each parameter includes an identifier, a data type, and optionally an `inputBinding` which describes how this input parameter should appear on the command line. In this example, the `position` field indicates where it should appear on the command line.

```
outputs: []
```

This tool has no formal output, so the `outputs` section is an empty list.

2.2 Essential input parameters

The `inputs` of a tool is a list of input parameters that control how to run the tool. Each parameter has an `id` for the name of parameter, and `type` describing what types of values are valid for that parameter.

Available primitive types are *string*, *int*, *long*, *float*, *double*, and *null*; complex types are *array* and *record*; in addition there are special types *File*, *Directory* and *Any*.

The following example demonstrates some input parameters with different types and appearing on the command line in different ways:

inp.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: echo
inputs:
  example_flag:
    type: boolean
    inputBinding:
      position: 1
      prefix: -f
  example_string:
    type: string
    inputBinding:
      position: 3
      prefix: --example-string
  example_int:
    type: int
    inputBinding:
      position: 2
      prefix: -i
      separate: false
  example_file:
    type: File?
    inputBinding:
      prefix: --file=
      separate: false
      position: 4

outputs: []
```

inp-job.yml

```
example_flag: true
example_string: hello
example_int: 42
example_file:
  class: File
  path: whale.txt
```

Notice that "example_file", as a `File` type, must be provided as an object with the fields `class: File` and `path` .

Next, create a whale.txt and invoke `cwl-runner` with the tool wrapper and the input object on the command line:

```
$ touch whale.txt
$ cwl-runner inp.cwl inp-job.yml
[job 140020149614160] /home/example$ echo -f -i42 --example-string hello --file=/home/example/whale.txt
-f -i42 --example-string hello --file=/home/example/whale.txt
Final process status is success
```

The field `inputBinding` is optional and indicates whether and how the input parameter should be appear on the tool's command line. If `inputBinding` is missing, the parameter does not appear on the command line. Let's look at each example in detail.

```
example_flag:
  type: boolean
  inputBinding:
    position: 1
    prefix: -f
```

Boolean types are treated as a flag. If the input parameter "example_flag" is "true", then `prefix` will be added to the command line. If false, no flag is added.

```
example_string:
  type: string
  inputBinding:
    position: 3
    prefix: --example-string
```

String types appear on the command line as literal values. The `prefix` is optional, if provided, it appears as a separate argument on the command line before the parameter . In the example above, this is rendered as `--example-string hello` .

```
example_int:
  type: int
  inputBinding:
    position: 2
    prefix: -i
    separate: false
```

Integer (and floating point) types appear on the command line with decimal text representation. When the option `separate` is false (the default value is true), the prefix and value are combined into a single argument. In the example above, this is rendered as `-i42` .

```
example_file:
  type: File?
  inputBinding:
    prefix: --file=
    separate: false
    position: 4
```

File types appear on the command line as the path to the file. When the parameter type ends with a question mark `?` it indicates that the parameter is optional. In the example above, this is rendered as `--file=/home/example/whale.txt` . However, if the "example_file" parameter were not provided in the input, nothing would appear on the command line.

Input files are read-only. If you wish to update an input file, you must first copy it to the output directory.

The value of `position` is used to determine where parameter should appear on the command line. Positions are relative to one another, not absolute. As a result, positions do not have to be sequential, three parameters with positions `[1, 3, 5]` will result in the same command line as `[1, 2, 3]` . More than one parameter can have the same position (ties are broken using the parameter name), and the position field itself is optional. the default position is 0.

The `baseCommand` field always comes before parameters.

2.3 Returning output files

The `outputs` of a tool is a list of output parameters that should be returned after running the tool. Each parameter has an `id` for the name of parameter, and `type` describing what types of values are valid for that parameter.

When a tool runs under CWL, the starting working directory is the designated output directory. The underlying tool or script must record its results in the form of files created in the output directory. The output parameters returned by the CWL tool are either the output files themselves, or come from examining the content of those files.

tar.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: [tar, xf]
inputs:
  tarfile:
    type: File
    inputBinding:
      position: 1
outputs:
  example_out:
    type: File
    outputBinding:
      glob: hello.txt
```

tar-job.yml

```
tarfile:
  class: File
  path: hello.tar
```

Next, create a tar file for the example and invoke `cwl-runner` with the tool wrapper and the input object on the command line:

```
$ touch hello.txt && tar -cvf hello.tar hello.txt
$ cwl-runner tar.cwl tar-job.yml
[job 139868145165200] $ tar xf /home/example/hello.tar
Final process status is success
{
"example_out": {
  "location": "hello.txt",
  "size": 13,
  "class": "File",
  "checksum": "sha1$47a013e660d408619d894b20806b1d5086aab03b"
  }
}
```

The field `outputBinding` describes how to to set the value of each output parameter.

```
outputs:
  example_out:
    type: File
    outputBinding:
      glob: hello.txt
```

The `glob` field consists of the name of a file in the output directory. If you don't know name of the file in advance, you can use a wildcard pattern.

2.4 Capturing a tool's standard output stream

To capture a tool's standard output stream, add the `stdout` field with the name of the file where the output stream should go. Then add `type: stdout` on the corresponding output parameter.

stdout.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: echo
stdout: output.txt
inputs:
  message:
    type: string
    inputBinding:
      position: 1
outputs:
  output:
    type: stdout
```

echo-job.yml

```
message: Hello world!
```

Now invoke `cwl-runner` providing the tool wrapper and the input object on the command line:

```
$ cwl-runner stdout.cwl echo-job.yml
[job 140199012414352] $ echo 'Hello world!' > output.txt
Final process status is success
{
"output": {
  "location": "output.txt",
  "size": 13,
  "class": "File",
  "checksum": "sha1$47a013e660d408619d894b20806b1d5086aab03b"
  }
}
$ cat output.txt
Hello world!
```

2.5 Parameter references

In a previous example, we extracted a file using the "tar" program. However, that example was very limited because it assumed that the file we were interested in was called "hello.txt". In this example, you will see how to reference the value of input parameters dynamically from other fields.

tar-param.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: [tar, xf]
inputs:
  tarfile:
    type: File
    inputBinding:
      position: 1
  extractfile:
    type: string
    inputBinding:
      position: 2
outputs:
  example_out:
    type: File
    outputBinding:
      glob: $(inputs.extractfile)
```

tar-param-job.yml

```
tarfile:
  class: File
  path: hello.tar
extractfile: goodbye.txt
```

Create your input files and invoke `cwl-runner` with the tool wrapper and the input object on the command line:

```
$ rm hello.tar || true && touch goodbye.txt && tar -cvf hello.tar goodbye.txt
$ cwl-runner tar-param.cwl tar-param-job.yml
[job 139868145165200] $ tar xf /home/example/hello.tar goodbye.txt
Final process status is success
{
"example_out": {
  "location": "goodbye.txt",
  "size": 24,
  "class": "File",
  "checksum": "sha1$dd0a4c4c49ba43004d6611771972b6cf969c1c01"
  }
}
```

Certain fields permit parameter references which are enclosed in `$(...)` . These are evaluated and replaced with value being referenced.

```
outputs:
  example_out:
    type: File
    outputBinding:
      glob: $(inputs.extractfile)
```

References are written using a subset of Javascript syntax. In this example, `$(inputs.extractfile)` , `$(inputs["extractfile"])` , and `$(inputs['extractfile'])` are equivalent.

The value of the "inputs" variable is the input object provided when the CWL tool was invoked.

Note that because File parameters are objects, to get the path to an input file you must reference the path field on a file object; to reference the path to the tar file in the above example you would write `$(inputs.tarfile.path)` .

2.6 Running tools inside Docker

Docker (http://docker.io) containers simplify software installation by providing a complete known-good runtime for software and its dependencies. However, containers are also purposefully isolated from the host system, so in order to run a tool inside a Docker container there is additional work to ensure that input files are available inside the container and output files can be recovered from the container. CWL can perform this work automatically, allowing you to use Docker to simplify your software management while avoiding the complexity of invoking and managing Docker containers.

This example runs a simple Node.js script inside a Docker container.

docker.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: node
hints:
  DockerRequirement:
    dockerPull: node:slim
inputs:
  src:
    type: File
    inputBinding:
      position: 1
outputs: []
```

docker-job.yml

```
src:
  class: File
  path: hello.js
```

Provide a hello.js and invoke `cwl-runner` providing the tool wrapper and the input object on the command line:

```
$ echo "console.log(\"Hello World\");" > hello.js
$ cwl-runner docker.cwl docker-job.yml
[job 140259721854416] /home/example$ docker run -i --volume=/home/example/hello.js:/var/lib/cwl/job369354770_examples/hello.js:ro --volu
me=/home/example:/var/spool/cwl:rw --volume=/tmp/tmpDLs5hm:/tmp:rw --workdir=/var/spool/cwl --read-only=true --net=none --user=1001 --rm
--env=TMPDIR=/tmp node:slim node /var/lib/cwl/job369354770_examples/hello.js
Hello world!
Final process status is success
```

Notice the CWL runner has constructed a Docker command line to run the script. One of the responsibilies of the CWL runner is to the paths of input files to reflect the location where they appear inside the container. In this example, the path to the script `hello.js` is `/home/example/hello.js` outside the container but `/var/lib/cwl/job369354770_examples/hello.js` inside the container, as reflected in the invocation of the `node` command.

2.7 Additional command line arguments and runtime parameters

Sometimes tools require additional command line options that don't correspond exactly to input parameters.

In this example, we will wrap the Java compiler to compile a java source file to a class file. By default, `javac` will create the class files in the same directory as the source file. However, CWL input files (and the directories in which they appear) may be read-only, so we need to instruct javac to write the class file to the designated output directory instead.

arguments.cwl

```
cwlVersion: v1.0
class: CommandLineTool
label: Example trivial wrapper for Java 7 compiler
hints:
  DockerRequirement:
    dockerPull: java:7-jdk
baseCommand: javac
arguments: ["-d", $(runtime.outdir)]
inputs:
  src:
    type: File
    inputBinding:
      position: 1
outputs:
  classfile:
    type: File
    outputBinding:
      glob: "*.class"
```

arguments-job.yml

```
src:
  class: File
  path: Hello.java
```

Now create a sample Java file and invoke `cwl-runner` providing the tool wrapper and the input object on the command line:

```
$ echo "public class Hello {}" > Hello.java
$ cwl-runner arguments.cwl arguments-job.yml
[job arguments.cwl] /tmp/tmpwYALo1$ docker \
 run \
 -i \
 --volume=/home/peter/work/common-workflow-language/v1.0/examples/Hello.java:/var/lib/cwl/stg8939ac04-7443-4990-a518-1855b2322141/Hello.
java:ro \
 --volume=/tmp/tmpwYALo1:/var/spool/cwl:rw \
 --volume=/tmp/tmpptIAJ8:/tmp:rw \
 --workdir=/var/spool/cwl \
 --read-only=true \
 --user=1001 \
 --rm \
 --env=TMPDIR=/tmp \
 --env=HOME=/var/spool/cwl \
 java:7 \
 javac \
 -d \
 /var/spool/cwl \
 /var/lib/cwl/stg8939ac04-7443-4990-a518-1855b2322141/Hello.java
Final process status is success
{
  "classfile": {
    "size": 416,
    "location": "/home/example/Hello.class",
    "checksum": "sha1$2f7ac33c1f3aac3f1fec7b936b6562422c85b38a",
    "class": "File"
  }
}
```

Here we use the `arguments` field to add an additional argument to the command line that isn't tied to a specific input parameter.

```
arguments: ["-d", $(runtime.outdir)]
```

This example references a runtime parameter. Runtime parameters provide information about the hardware or software environment when the tool is actually executed. The `$(runtime.outdir)` parameter is the path to the designated output directory. Other parameters include `$(runtime.tmpdir)`, `$(runtime.ram)`, `$(runtime.cores)`, `$(runtime.outdirSize)`, and `$(runtime.tmpdirSize)`. See the Runtime Environment (CommandLineTool.html#Runtime_environment) section of the CWL specification for details.

2.8 Array inputs

It is easy to add arrays of input parameters represented to the command line. To specify an array parameter, the array definition is nested under the `type` field with `type: array` and `items` defining the valid data types that may appear in the array.

array-inputs.cwl

```
cwlVersion: v1.0
class: CommandLineTool
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

outputs: []
baseCommand: echo
```

array-inputs-job.yml

```
filesA: [one, two, three]
filesB: [four, five, six]
filesC: [seven, eight, nine]
```

Now invoke `cwl-runner` providing the tool wrapper and the input object on the command line:

```
$ cwl-runner array-inputs.cwl array-inputs-job.yml
[job 140334923640912] /home/example$ echo -A one two three -B=four -B=five -B=six -C=seven,eight,nine
-A one two three -B=four -B=five -B=six -C=seven,eight,nine
Final process status is success
{}
```

The `inputBinding` can appear either on the outer array parameter definition or the inner array element definition, and these produce different behavior when constructing the command line, as shown above. In addition, the `itemSeperator` field, if provided, specifies that array values should be concatenated into a single argument separated by the item separator string.

You can specify arrays of arrays, arrays of records, and other complex types.

2.9 Array outputs

You can also capture multiple output files into an array of files using `glob` .

array-outputs.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: touch
inputs:
  touchfiles:
    type:
      type: array
      items: string
    inputBinding:
      position: 1
outputs:
  output:
    type:
      type: array
      items: File
    outputBinding:
      glob: "*.txt"
```

array-outputs-job.yml

```
touchfiles:
  - foo.txt
  - bar.dat
  - baz.txt
```

Now invoke `cwl-runner` providing the tool wrapper and the input object on the command line:

```
$ cwl-runner array-outputs.cwl array-outputs-job.yml
[job 140190876078160] /home/example$ touch foo.txt bar.dat baz.txt
Final process status is success
{
  "output": [
    {
      "size": 0,
      "location": "/home/peter/work/common-workflow-language/draft-3/examples/foo.txt",
      "checksum": "sha1$da39a3ee5e6b4b0d3255bfef95601890afd80709",
      "class": "File"
    },
    {
      "size": 0,
      "location": "/home/peter/work/common-workflow-language/draft-3/examples/baz.txt",
      "checksum": "sha1$da39a3ee5e6b4b0d3255bfef95601890afd80709",
      "class": "File"
    }
  ]
}
```

2.10 Record inputs, dependent and mutually exclusive parameters

Sometimes an underlying tool has several arguments that must be provided together (they are dependent) or several arguments that cannot be provided together (they are exclusive). You can use records and type unions to group parameters together to describe these two conditions.

record.cwl

```
cwlVersion: v1.0
class: CommandLineTool
inputs:
  dependent_parameters:
    type:
      type: record
      name: dependent_parameters
      fields:
        itemA:
          type: string
          inputBinding:
            prefix: -A
        itemB:
          type: string
          inputBinding:
            prefix: -B
  exclusive_parameters:
    type:
      - type: record
        name: itemC
        fields:
          itemC:
            type: string
            inputBinding:
              prefix: -C
      - type: record
        name: itemD
        fields:
          itemD:
            type: string
            inputBinding:
              prefix: -D
outputs: []
baseCommand: echo
```

record-job1.yml

```
dependent_parameters:
  itemA: one
exclusive_parameters:
  itemC: three
```

```
$ cwl-runner record.cwl record-job1.yml
Workflow error:
  Error validating input record, could not validate field `dependent_parameters` because
  missing required field `itemB`
```

In the first example, you can't provide `itemA` without also providing `itemB` .

record-job2.yml

```
dependent_parameters:
  itemA: one
  itemB: two
exclusive_parameters:
  itemC: three
  itemD: four
```

```
$ cwl-runner record.cwl record-job2.yml
[job 140566927111376] /home/example$ echo -A one -B two -C three
-A one -B two -C three
Final process status is success
{}
```

In the second example, `itemC` and `itemD` are exclusive, so only `itemC` is added to the command line and `itemD` is ignored.

record-job3.yml

```
dependent_parameters:
  itemA: one
  itemB: two
exclusive_parameters:
  itemD: four
```

```
$ cwl-runner record.cwl record-job3.yml
[job 140606932172880] /home/example$ echo -A one -B two -D four
-A one -B two -D four
Final process status is success
{}
```

In the third example, only `itemD` is provided, so it appears on the command line.

2.11 Environment variables

Tools run in a restricted environment and do not inherit most environment variables from the parent process. You can set environment variables for the tool using `EnvVarRequirement` .

env.cwl

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: env
requirements:
  EnvVarRequirement:
    envDef:
      HELLO: $(inputs.message)
inputs:
  message: string
outputs: []
```

echo-job.yml

```
message: Hello world!
```

Now invoke `cwl-runner` with the tool wrapper and the input object on the command line:

```
$ cwl-runner env.cwl echo-job.yml
[job 140710387785808] /home/example$ env
PATH=/bin:/usr/bin:/usr/local/bin
HELLO=Hello world!
TMPDIR=/tmp/tmp63Obpk
Final process status is success
{}
```

2.12 Javascript expressions

If you need to manipulate input parameters, include the requirement `InlineJavascriptRequirement` and then anywhere a parameter reference is legal you can provide a fragment of Javascript that will be evaluated by the CWL runner.

expression.cwl

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

requirements:
  - class: InlineJavascriptRequirement

inputs: []
outputs: []
arguments:
  - prefix: -A
    valueFrom: $(1+1)
  - prefix: -B
    valueFrom: $("/foo/bar/baz".split('/').slice(-1)[0])
  - prefix: -C
    valueFrom: |
      ${
        var r = [];
        for (var i = 10; i >= 1; i--) {
          r.push(i);
        }
        return r;
      }
```

As this tool does not require any `inputs` we can run it with an (almost) empty job file:

empty.yml

```
{}
|
```

We can then run `expression.cwl` :

```
$ cwl-runner expression.cwl empty.yml
[job 140000594593168] /home/example$ echo -A 2 -B baz -C 10 9 8 7 6 5 4 3 2 1
-A 2 -B baz -C 10 9 8 7 6 5 4 3 2 1
Final process status is success
{}
```

You can only use expressions in certain fields. These are: `filename` , `fileContent` , `envValue` , `valueFrom` , `glob` , `outputEval` , `stdin` , `stdout` , `coresMin` , `coresMax` , `ramMin` , `ramMax` , `tmpdirMin` , `tmpdirMax` , `outdirMin` , and `outdirMax` .

2.13 Creating files at runtime

Sometimes you need to create a file on the fly from input parameters, such as tools which expect to read their input configuration from a file rather than the command line parameters. To do this, use `InitialWorkDirRequirement` .

createfile.cwl

```
class: CommandLineTool
cwlVersion: v1.0
baseCommand: ["cat", "example.conf"]

requirements:
  InitialWorkDirRequirement:
    listing:
      - entryname: example.conf
        entry: |
          CONFIGVAR=$(inputs.message)

inputs:
  message: string
outputs: []
```

echo-job.yml

```
message: Hello world!
```

Now invoke `cwl-runner` with the tool wrapper and the input object on the command line:

```
$ cwltool createfile.cwl echo-job.yml
[job 140528604979344] /home/example$ cat example.conf
CONFIGVAR=Hello world!
Final process status is success
{}
```

2.14 Staging input files in the output directory

Normally, input files are located in a read-only directory separate from the output directory. This causes problems if the underlying tool expects to write its output files alongside the input file in the same directory. You use `InitialWorkDirRequirement` to stage input files into the output directory. In this example, we use a Javascript expression to extract the base name of the input file from its leading directory path.

linkfile.cwl

```
cwlVersion: v1.0
class: CommandLineTool
hints:
  DockerRequirement:
    dockerPull: java:7
baseCommand: javac

requirements:
  - class: InlineJavascriptRequirement
  - class: InitialWorkDirRequirement
    listing:
      - $(inputs.src)

inputs:
  src:
    type: File
    inputBinding:
      position: 1
      valueFrom: $(self.basename)

outputs:
  classfile:
    type: File
    outputBinding:
      glob: "*.class"
```

arguments-job.yml

```
src:
  class: File
  path: Hello.java
```

Now invoke `cwl-runner` with the tool wrapper and the input object on the command line:

```
$ cwl-runner linkfile.cwl arguments-job.yml
[job 139928309171664] /home/example$ docker run -i --volume=/home/example/Hello.java:/var/lib/cwl/job557617295_examples/Hello.java:ro --
volume=/home/example:/var/spool/cwl:rw --volume=/tmp/tmpmNbApw:/tmp:rw --workdir=/var/spool/cwl --read-only=true --net=none --user=1001
 --rm --env=TMPDIR=/tmp java:7 javac Hello.java
Final process status is success
{
"classfile": {
  "size": 416,
  "location": "/home/example/Hello.class",
  "checksum": "sha1$2f7ac33c1f3aac3f1fec7b936b6562422c85b38a",
  "class": "File"
  }
}
```

3. Writing Workflows

3.1 First workflow

This workflow extracts a java source file from a tar file and then compiles it.

1st-workflow.cwl

```
cwlVersion: v1.0
class: Workflow
inputs:
  inp: File
  ex: string

outputs:
  classout:
    type: File
    outputSource: compile/classfile

steps:
  untar:
    run: tar-param.cwl
    in:
      tarfile: inp
      extractfile: ex
    out: [example_out]

  compile:
    run: arguments.cwl
    in:
      src: untar/example_out
    out: [classfile]
```

Use a JSON object in a separate file to describe the input of a run:

1st-workflow-job.yml

```
inp:
  class: File
  path: hello.tar
ex: Hello.java
```

Now invoke `cwl-runner` with the tool wrapper and the input object on the command line:

```
$ echo "public class Hello {}" > Hello.java && tar -cvf hello.tar Hello.java
$ cwl-runner 1st-workflow.cwl 1st-workflow-job.yml
[job untar] /tmp/tmp94qFiM$ tar xf /home/example/hello.tar Hello.java
[step untar] completion status is success
[job compile] /tmp/tmpu1iaKL$ docker run -i --volume=/tmp/tmp94qFiM/Hello.java:/var/lib/cwl/job301600808_tmp94qFiM/Hello.java:ro --volum
e=/tmp/tmpu1iaKL:/var/spool/cwl:rw --volume=/tmp/tmpfZnNdR:/tmp:rw --workdir=/var/spool/cwl --read-only=true --net=none --user=1001 --rm
--env=TMPDIR=/tmp java:7 javac -d /var/spool/cwl /var/lib/cwl/job301600808_tmp94qFiM/Hello.java
[step compile] completion status is success
[workflow 1st-workflow.cwl] outdir is /home/example
Final process status is success
{
  "classout": {
    "location": "/home/example/Hello.class",
    "checksum": "sha1$e68df795c0686e9aa1a1195536bd900f5f417b18",
    "class": "File",
    "size": 416
  }
}
```

What's going on here? Let's break it down:

```
cwlVersion: v1.0
class: Workflow
```

The `cwlVersion` field indicates the version of the CWL spec used by the document. The `class` field indicates this document describes a workflow.

```
inputs:
  inp: File
  ex: string
```

The `inputs` section describes the inputs of the workflow. This is a list of input parameters where each parameter consists of an identifier and a data type. These parameters can be used as sources for input to specific workflows steps.

```
outputs:
  classout:
    type: File
    outputSource: compile/classfile
```

The `outputs` section describes the outputs of the workflow. This is a list of output parameters where each parameter consists of an identifier and a data type. The `outputSource` connects the output parameter `classfile` of the `compile` step to the workflow output parameter `classout` .

```
steps:
  untar:
    run: tar-param.cwl
    in:
      tarfile: inp
      extractfile: ex
    outputs: [example_out]
```

The `steps` section describes the actual steps of the workflow. In this example, the first step extracts a file from a tar file, and the second step compiles the file from the first step using the java compiler. Workflow steps are not necessarily run in the order they are listed, instead the order is determined by the dependencies between steps (using `source`). In addition, workflow steps which do not depend on one another may run in parallel.

The first step, `untar` runs `tar-param.cwl` (described previously in Parameter references). This tool has two input parameters, `tarfile` and `extractfile` and one output parameter `example_out` .

The `inputs` section of the workflow step connects these two input parameters to the inputs of the workflow, `inp` and `ex` using `source` . This means that when the workflow step is executed, the values assigned to `inp` and `ex` will be used for the parameters `tarfile` and `extractfile` in order to run the tool.

The `outputs` section of the workflow step lists the output parameters that are expected from the tool.

```
  compile:
    run: arguments.cwl
    in:
      src: untar/example_out
    outputs: [classfile]
```

The second step `compile` depends on the results from the first step by connecting the input parameter `src` to the output parameter of `untar` using `untar/example_out` . The output of this step `classfile` is connected to the `outputs` section for the Workflow, described above.

3.2 Nested workflows

Workflows are ways to combine multiple tools to perform a larger operations. We can also think of a workflow as being a tool itself; a CWL workflow can be used as a step in another CWL workflow, if the workflow engine supports the `SubworkflowFeatureRequirement` :

```
requirements:
  - class: SubworkflowFeatureRequirement
```

Here's an example workflow that uses our `1st-workflow.cwl` as a nested workflow:

```
cwlVersion: v1.0
class: Workflow

inputs: []

outputs:
  classout:
    type: File
    outputSource: compile/classout

requirements:
  - class: SubworkflowFeatureRequirement

steps:
  compile:
    run: 1st-workflow.cwl
    in:
      inp:
        source: create-tar/tar
      ex:
        default: "Hello.java"
    out: [classout]

  create-tar:
    requirements:
      - class: InitialWorkDirRequirement
        listing:
          - entryname: Hello.java
            entry: |
              public class Hello {
                public static void main(String[] argv) {
                    System.out.println("Hello from Java");
                }
              }
    in: []
    out: [tar]
    run:
      class: CommandLineTool
      requirements:
        - class: ShellCommandRequirement
      arguments:
        - shellQuote: false
          valueFrom: |
            date
            tar cf hello.tar Hello.java
            date
      inputs: []
      outputs:
        tar:
          type: File
          outputBinding:
            glob: "hello.tar"
```

A CWL `Workflow` can be used as a `step` just like a `CommandLineTool`, it's CWL file is included with `run`. The workflow inputs (`inp` and `ex`) and outputs (`classout`) then can be mapped to become the step's input/outputs.

```
  compile:
    run: 1st-workflow.cwl
    in:
      inp:
        source: create-tar/tar
      ex:
        default: "Hello.java"
    out: [classout]
```

Our `1st-workflow.cwl` was parameterized with workflow inputs, so when running it we had to provide a job file to denote the tar file and `*.java` filename. This is generally best-practice, as it means it can be reused in multiple parent workflows, or even in multiple steps within the same workflow.

Here we use `default:` to hard-code `"Hello.java"` as the `ex` input, however our workflow also requires a tar file at `inp`, which we will prepare in the `create-tar` step. At this point it is probably a good idea to refactor `1st-workflow.cwl` to have more specific input/output names, as those also appear in its usage as a tool.

It is also possible to do a less generic approach and avoid external dependencies in the job file. So in this workflow we can generate a hard-coded `Hello.java` file using the previously mentioned `InitialWorkDirRequirement` requirement, before adding it to a tar file.

```
  create-tar:
    requirements:
      - class: InitialWorkDirRequirement
        listing:
          - entryname: Hello.java
            entry: |
              public class Hello {
                public static void main(String[] argv) {
                    System.out.println("Hello from Java");
                }
              }
```

In this case our step can assume `Hello.java` rather than be parameterized, so we can use a simpler `arguments` form as long as the CWL workflow wngine supports the `ShellCommandRequirement`:

```
run:
  class: CommandLineTool
  requirements:
    - class: ShellCommandRequirement
  arguments:
    - shellQuote: false
      valueFrom: >
        tar cf hello.tar Hello.java
```

Note the use of `shellQuote: false` here, otherwise the shell will try to execute the quoted binary `"tar cf hello.tar Hello.java"` .

Here the `>` block means that newlines are stripped, so it's possible to write the single command on multiple lines. Similarly, the `|` we used above will preserve newlines, combined with `ShellCommandRequirement` this would allow embedding a shell script. Shell commands should however be used sparingly in CWL, as it means you "jump out" of the workflow and no longer get reusable components, provenance or scalability. For reproducibility and portability it is recommended to only use shell commands together with a `DockerRequirement` hint, so that the commands are executed in a predictable shell environment.

Did you notice that we didn't split out the `tar cf` tool to a separate file, but rather embedded it within the CWL Workflow file? This is generally not best practice, as the tool then can't be reused. The reason for doing it in this case is because the command line is hard-coded with filenames that only make sense within this workflow.

In this example we had to prepare a tar file outside, but only because our inner workflow was designed to take that as an input. A better refactoring of the inner workflow would be to take a list of Java files to compile, which would simplify its usage as a tool step in other workflows.

Nested workflows can be a powerful feature to generate higher-level functional and reusable workflow units - but just like for creating a CWL Tool description, care must be taken to improve its usability in multiple workflows.