Sage Workflows User Guide

Release 0.0.1

Jun 18, 2020

Overview

Numerous groups are developing technologies and best practices for describing and running genomic analyses in a portable and reproducible fashion. Key among these technologies are specifications to describe workflows and tools/tasks such as the Common Workflow Language (CWL) and the Workflow Description Language (WDL) as well as software containers such as Docker. These enable scientists to express and run complex workflows by explicitly defining the inputs and outputs of tools/tasks, how the outputs of tools are passed to the inputs of others, and runtime requirements.

Workflow execution should support multiple tasks and domains at Sage, including challenge infrastructure, data processing pipelines, computationally intensive analyses, and benchmarking for scientific communities. The WG will support teams by developing scalable, documented workflows, as well as the systems and guidance to author, test, execute, and share these workflows for diverse applications. Our goal is to prototype and demonstrate solutions, and define requirements and specifications for the Synapse platform team. Members of this team will participate in and monitor external groups and communities — to stay aware of and informed on best practices and emerging standards. Any systems we devise and implement should conform to and enable FAIR principles; in particular, tight integration with data provenance in Synapse should be a long term goal of any workflow-related developments.

CHAPTER 1

Rationale

1.1 Why use workflows?

Software container technologies, such as Docker, allow researchers to create lightweight virtual machines into which they can install software, configuration, and small data files. The resulting images can be shared across a variety of cloud or local computing platforms. Such "Dockerized" algorithms can be "moved" to the data (i.e., uploaded in the same computer system containing the data) in contrast to the traditional format of "moving data" to modelers (i.e., modelers download data to their computational space, which could be onerous when datasets are massive). One advantage to this framework are that it facilitates the use of protected, proprietary, and very large datasets — with vastly reduced data-transfer, improved data security, and reduction of administrative and IRB overhead. Another advantage is the creation of a library of well-annotated algorithms with consistent input and output parameters for community use.

While container technologies provide a reliable and interoperable way to move models across computational environments, orchestration of large numbers of containers and distributed data assets requires significant engineering overhead. Workflow execution systems orchestrate one or more asynchronous tasks, linked by dependency relationships. Managing container execution through hardened workflow systems (e.g., Toil, FireCloud/Cromwell, Galaxy) offers several benefits: (i) Scalability: built-in support for resource provisioning, distributed task execution, autoscaling, and utilization of different backends for scheduling and load balancing; (ii) Portability: coordination of dependencies and data among containerized steps for reproducible computing across environments; (iii) Reentrancy: the ability of a program to continue where it left off if interrupted through sophisticated caching mechanisms and failure/recovery logic.

CHAPTER 2

Workflows 101

CHAPTER 3

Workflow Standards

CHAPTER 4

Workflow Engines

There are a diverse number of workflow engines and languages available. This figure below illustrates the compatibility between the different engines and languages.

Pending
In development
Dependent on language support
Rumored

	Can run workflows written in:		*Can find/share workflows written in:*	*Can run remote jobs for workflows written in:*
Engine/platform	**CWL**	**WDL**	**TRS**	**WES**
Toil	x	x	CWL, WDL	CWL, WDL
Arvados	x		CWL	CWL
Cromwell	x	x	CWL, WDL	CWL, WDL
Airflow	x		CWL	
Nextflow	x		CWL, NF	
Snakemake	x		CWL	
Galaxy (via Planemo)	x		CWL	
FireCloud		x	CWL, WDL	CWL, WDL
DNAnexus	x	x	CWL, WDL	WDL
DNAstack		x	WDL	
Seven Bridges	x		CWL	CWL

4.1 Common Workflow Language (CWL)

- As an organization we are working to provide Synapse support to CWL primarily

General rationale is documented below:

- CWL is literally a community-driven standard (two of its main "competitors" — Jeff Gentry / WDL and John Chilton / Galaxy — are on the leadership team). not only was it designed as a standard, but it was designed first and foremost to ensure portability and reproducibility — something that's fairly important in our field
- there are multiple implementations of CWL (in this case, by implementations I mean execution engines or platforms that support CWL), meaning one can take a CWL tool/workflow and run it not just in different environments, but using different software. this cannot be said for Nextflow, Snakemake, Galaxy, or even WDL. if the maintainers of those platforms decide to kill the project (unlikely I know), then your workflow now has a much more finite lifespan. conversely, each of the the platforms for those languages — as mentioned — are working on support for CWL, even if it's just at the stage of import/export

- while CWL can be harder to use and might not be quite as expressive or powerful as Nextflow and Snakemake.. it's still pretty damn powerful. i also think it provides a particularly nice interface for Docker images, which in a lot of cases can just end up as black boxes
- CWL as a language is built on concepts related to linked data, and it provides far more support for semantics, annotation, validation of inputs/outputs, resolution of paths and locations, etc. than other platforms. I think some of these features are underutilized in most real world examples, but could be super useful if fully harnessed
- there are already multiple groups thinking about (and working implementations of) combining CWL with provenance standards (i.e., W3C PROV)
- there's a related (still pending official approval GA4GH) API called the Tool Registry Service (TRS) that allows users to describe, share, and find tools/workflows in a standardized way. TRS and it's main implementation, Dockstore, historically supported CWL and WDL, though now can support Nextflow as well; the folks at Biocontainers just started to roll out their own implementation of TRS, which will allow users to more easily connect their Docker images and the workflows that use them

CHAPTER 5

Getting Started with CWL

CHAPTER 6

CWL Inputs & Outputs

CHAPTER 7

Using Containers

CHAPTER 8

JavaScript & CWL

8.1 Part 1: InlineJavascriptRequirement

If you need to do a computation while running a cwltool, you can do so using a snippet of javascript.

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

$namespaces:
  s: https://schema.org/

s:author:
  - class: s:Person
    s:identifier: https://orcid.org/0000-0002-0326-7494
    s:email: andrew.lamb@sagebase.org

s:name: Andrew Lamb

cwlVersion: v1.0

class: CommandLineTool

requirements:
- class: InlineJavascriptRequirement
- class: InitialWorkDirRequirement
  listing:
  - entry: $(inputs.file)
    writable: true

baseCommand:
- gzip

inputs:

- id: file
```

(continues on next page)

(continued from previous page)

```
  type: File
  inputBinding:
    position: 1

outputs:

- id: gziped_file
  type: File
  outputBinding:
    glob: $(inputs.file.path + ".gz")
```

The above tool gzips the given input file. The gzip util will tack on the the .gz suffix, so we don't know what the exact file name will be. But we can figure it out using a little bit of javascript:

```
- id: gziped_file
  type: File
  outputBinding:
    glob: $(inputs.file.path + ".gz")
```

1. Inputs.file.path returns the path of the input file
2. '+ ".gz"' concatenates the gz suffix
3. $() returns the result of the javscript expression contained between the parens

8.2 Part 2: Expression tools

Expression tools are cwltools that only perform javascript, and don't call any other script or command.

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

$namespaces:
  s: https://schema.org/

s:author:
  - class: s:Person
    s:identifier: https://orcid.org/0000-0002-0326-7494
    s:email: andrew.lamb@sagebase.org

s:name: Andrew Lamb

cwlVersion: v1.0
class: ExpressionTool

requirements:
- class: InlineJavascriptRequirement

inputs:

- id: input_file
  type: File
- id: new_file_name
  type: string

outputs:
```

(continues on next page)

(continued from previous page)

```
- id: output_file
  type: File

expression: |
  ${
    inputs.input_file.basename = inputs.new_file_name;
    return {output_file: inputs.input_file};
  }
```

Expression tools are like command line tools in terms of input and outputs. The difference is that instead of execution a commnad, expression tools execute a javascript expression:

```
expression: |
  ${
    inputs.input_file.basename = inputs.new_file_name;
    return {output_file: inputs.input_file};
  }
```

This expression simply renames the file, and returns it.

CHAPTER 9

Scattering Inputs

9.1 Part 1: Getting started

If you need to run a tool or workflow on an array, or multiple arrays of inputs, scatter is the way to accomplish this. We will be using the following tool as example of what we are looping over:

This runs the linux command wc(word count) on an input file, with the option to use the -l(lines) flag. Let's assume we want to run this tool on an array of files. (You can use the wc command on a list of files, but let's ignore this for the example.)

The way to run a tool on an array of inputs is to do it at the workflow level:

Let's go through the relevant parts.

This is necessary for using the scatter functionality:

```
requirements:
- class: ScatterFeatureRequirement
```

We want to run the tool on a list of input files. This is indicated by placing square brackets after the type:

```
inputs:

  lines: boolean?tep
  file_array: File[]
```

We will get back an array of files. Note that the scatter step will always result in an array output of whatever type the you are scattering produces. For example if the tool produces a File, the scattered version will produce and array of files. If the tool produces an array, the scattered version produces an array of arrays. This is true if the output of the step is the final workflow output, as in the above example, or it's being fed into another step.

```
outputs:

  output_array:
    type: File[]
```

(continues on next page)

(continued from previous page)

```
    outputSource:
    - wc/output
```

Finally we need to specify where an what we are scattering

In this example we want to run the wc.cwl tool over multiple files. The tool only takes in one file, so we have to make the workflow run the tool multiple times. The tool has the file input named 'file', whereas the workflow has the array input named 'file_array'. If we gave the tool the array input here, normally this would cause an error since a file array is not the same as a file:

```
in:
  lines: lines
  file: file_array
```

However by adding the scatter definition, we are telling the workflow to iterate over the array of files, running the tool once per each item in the array:

```
scatter: file
```

Note that the item we scatter is the name of the tool input name, NOT the workflow input name.

9.2 Part 2: dotproduct

This is a continuation from part 1. We will also be using the wc.cwl tool from that example.

In part 1 we covered how to do a sample scatter on an array of files. We'll now extend that any number of arrays. When you want to scatter over multiple arrays, you will need to tell CWL how to handle that. For this example we will use the scatter method called "dotproduct".

You can use the dotproduct as long as the arrays are the same length. The length of the arrays will determine how time your tool is run, and thus the length of the output array. For example if you have two arrays of three items each, and both are scattered, the tool would be run three times, the first instance would take the first item from each array as parameters, the second instance would use the second item from each array, and so on. Lets see an example:

```
#!/usr/bin/env cwl-runner
#
# Authors: Andrew Lamb

cwlVersion: v1.0
class: Workflow

requirements:
- class: ScatterFeatureRequirement

inputs:

  line_array: boolean[]
  file_array: File[]


outputs:

  output_array:
    type: File[]
    outputSource:
```

(continues on next page)

(continued from previous page)

```
    - wc/output

steps:

  wc:
    run: wc.cwl
    in:
      lines: line_array
      file: file_array
    scatter:
      - lines
      - file
    scatterMethod: dotproduct
    out:
      - output
```

This is very similar to the first example, let's look at what's changed.

We are still iterating over an array of input files, but here we want to also control whether or not we use the lines flag or not, so we are now providing an array of booleans:

```
inputs:

  line_array: boolean[]
  file_array: File[]
```

We now need to scatter two array inputs:

```
scatter:
  - lines
  - file
```

Finally since we are scattering more than one array we need to provide the method:

```
scatterMethod: dotproduct
```

9.3 Part 3: flat_crossproduct

This is a continuation from part 1 and 2. We will also be using the wc.cwl tool from part1

In part 1 we covered how to do a sample scatter on an array of files. In part 2 we extended that any number of arrays using the dotproduct. We will now look at scattering over multiple arrays using the flat crossproduct. Where the dotproduct required that your arrays be the same length, the flat crossproduct can scatter over arrays of different length. In addition, where the dotproduct result output is equal to that length of the arrays, the flat crossproduct result output is equal to: len(array1) * len(array2) * . . . len(array_n).

Another way of describing this is that the cwltool is run on every combination of inputs from each array. For example if you have an array of 3 files, and array of 2 flags, you will have 6 outputs. Each file will be run, once per each flag. The example workflow is exactly the same as the one in part2 except:

```
scatterMethod: flat_crossproduct
```

And the input yaml:

```
line_array:
- true
- false

file_array:
- class: File
  path: test_file1
- class: File
  path: test_file2
- class: File
  path: test_file3
```

And finally the output of "cwltool wc_workflow3.cwl wc_workflow.yaml" :


```
{
    "output_array": [
        {
            "path": "/home/aelamb/cwl_stuff/output.txt",
            "basename": "output.txt",
            "size": 70,
            "location": "file:///home/aelamb/cwl_stuff/output.txt",
            "class": "File",
            "checksum": "sha1$a912a8cf6107efe1bff86c42b7899e0a090d383c"
        },
        {
            "path": "/home/aelamb/cwl_stuff/output.txt",
            "basename": "output.txt",
            "size": 70,
            "location": "file:///home/aelamb/cwl_stuff/output.txt",
            "class": "File",
            "checksum": "sha1$ad06722d0c3641f8baf46242fcea51b77ee558e9"
        },
        {
            "path": "/home/aelamb/cwl_stuff/output.txt",
            "basename": "output.txt",
            "size": 70,
            "location": "file:///home/aelamb/cwl_stuff/output.txt",
            "class": "File",
            "checksum": "sha1$35470ddb936f3d1d3a5b907ff73c61d8df35d968"
        },
        {
            "path": "/home/aelamb/cwl_stuff/output.txt",
            "basename": "output.txt",
            "size": 74,
            "location": "file:///home/aelamb/cwl_stuff/output.txt",
            "class": "File",
            "checksum": "sha1$16fb2f95337e0b7c2b0e5076dc09b6509a762482"
        },
        {
            "path": "/home/aelamb/cwl_stuff/output.txt",
            "basename": "output.txt",
            "size": 77,
            "location": "file:///home/aelamb/cwl_stuff/output.txt",
            "class": "File",
            "checksum": "sha1$c5c3a3c1ff8ef9d4573f8238cb67c355225775d7"
        },
        {
```


(continues on next page)

(continued from previous page)

```
            "path": "/home/aelamb/cwl_stuff/output.txt",
            "basename": "output.txt",
            "size": 77,
            "location": "file:///home/aelamb/cwl_stuff/output.txt",
            "class": "File",
            "checksum": "sha1$b0fb51fac542b2b9f64d1408acabcfb61b8a4055"
        }
    ]
}
```

9.4 Part 4: nested_crossproduct

This is very similar to flat_crossproduct. The difference is that instead of one long flat array, you will receive a nested array as output:

```
#!/usr/bin/env cwl-runner
#
# Authors: Andrew Lamb

cwlVersion: v1.0
class: Workflow

requirements:
- class: ScatterFeatureRequirement

inputs:

  line_array: boolean[]
  file_array: File[]


outputs:

  output_array:
    type:
      type: array
      items:
        type: array
        items: File
    outputSource:
    - wc/output

steps:

  wc:
    run: wc.cwl
    in:
      lines: line_array
      file: file_array
    scatter:
      - lines
      - file
    scatterMethod: nested_crossproduct
    out:
```

(continues on next page)

(continued from previous page)

```
    - output
```

The output will look like:


```
{
    "output_array": [
        [
            {
                "location": "file:///home/aelamb/cwl_stuff/output.txt",
                "basename": "output.txt",
                "size": 70,
                "checksum": "sha1$e211886d70dfff0eb61fc917d75f184ce8b609b7",
                "class": "File",
                "path": "/home/aelamb/cwl_stuff/output.txt"
            },
            {
                "location": "file:///home/aelamb/cwl_stuff/output.txt",
                "basename": "output.txt",
                "size": 70,
                "checksum": "sha1$5a30593e67cc7d8e446b0ea1559da74fb35be45a",
                "class": "File",
                "path": "/home/aelamb/cwl_stuff/output.txt"
            },
            {
                "location": "file:///home/aelamb/cwl_stuff/output.txt",
                "basename": "output.txt",
                "size": 70,
                "checksum": "sha1$0220442cc49f0a4b3f82821725b40449c4e150f6",
                "class": "File",
                "path": "/home/aelamb/cwl_stuff/output.txt"
            }
        ],
        [
            {
                "location": "file:///home/aelamb/cwl_stuff/output.txt",
                "basename": "output.txt",
                "size": 74,
                "checksum": "sha1$ff65542777206d16635fa2c1a3e0e6376ea02a29",
                "class": "File",
                "path": "/home/aelamb/cwl_stuff/output.txt"
            },
            {
                "location": "file:///home/aelamb/cwl_stuff/output.txt",
                "basename": "output.txt",
                "size": 77,
                "checksum": "sha1$c5f042720e1f9e6cf75de5659ef01f547cd1d38f",
                "class": "File",
                "path": "/home/aelamb/cwl_stuff/output.txt"
            },
            {
                "location": "file:///home/aelamb/cwl_stuff/output.txt",
                "basename": "output.txt",
                "size": 77,
                "checksum": "sha1$e125e09c3b8a7d398014e791698dda762afb0bea",
                "class": "File",
                "path": "/home/aelamb/cwl_stuff/output.txt"
            }
```


(continues on next page)

(continued from previous page)

```
            ]
        ]
}
```

CHAPTER 10

Staging Folders

CWL stages all input files and directories in a random read-only temp directory away from the working directory. If these are part of the basecommand, arguments, or have an inputBinding, CWL will handle the file paths for you. However you the above limitations will not work in the following situations:

1. The File or Directory needs to be staged in the Docker image, but is not part of the command.
2. You need to change the file or directory in some way.
3. You need the file or directory to be in the working directory.
4. The path to the file or directory needs to be predictable.

10.1 Part 1: Input files in the working directory

Here is a python script that can use the sync to synapse function:

```
import synapseclient
import argparse
import synapseutils

if __name__ == '__main__':

    parser = argparse.ArgumentParser("Stores files in Synapse")

    parser.add_argument(
            '-m',
            '--manifest_file',
            type = str,
            required=True)

    parser.add_argument(
            '-c',
            '--synapse_config_file',
```

(continues on next page)

(continued from previous page)

```
            type = str,
            required=True)

    args = parser.parse_args()

    syn = synapseclient.Synapse(configPath=args.synapse_config_file)
    syn.login()

    synapseutils.sync.syncToSynapse(syn, args.manifest_file)
```

And here is the CWL tool that calls it:

```
#!/usr/bin/env cwl-runner
#
# Authors: Andrew Lamb

cwlVersion: v1.0
class: CommandLineTool

requirements:
- class: InitialWorkDirRequirement
  listing: $(inputs.files)

hints:
  DockerRequirement:
    dockerPull: quay.io/andrewelamb/python_synapse_client

baseCommand:
- python3
- /usr/local/bin/sync_to_synapse.py

inputs:

  files: File[]

  synapse_config_file:
    type: File
    inputBinding:
      prefix: "--synapse_config_file"

  manifest_file:
    type: File
    inputBinding:
      prefix: "--manifest_file"



outputs: []
```

In the above example we need to pass a manifest to the tool that has the paths of the files to be uploaded. If we didn't stage the files in the working directory, CWL would put them all in their own randomly generated temp directories. By placing them in the working directory we know that the relative paths will be just the name of the file.

To stage the files specified in the input files parameter we include the following:

```
requirements:
- class: InitialWorkDirRequirement
```

(continues on next page)

(continued from previous page)

```
  listing: $(inputs.files)
```

Notice that the below input does not have an inputBinding. This means its a parameter of the tool, but not the command the tool is constructing. This allows the file parameter to be referenced by the InitialWorkDirRequirement:

10.2 Part 2: Creating a config file in the working directory

The below tool needs a config file, where the last line is a directory that is being passed in an input. The directory will be put in a random location in the docker image, so the config file cannot be passed in as an input as well, but needs to be written after the path to the directory is known.

```
baseCommand: run-pipe

arguments:
- --config
- config_drops.ini

requirements:
  - class: InlineJavascriptRequirement
  - class: InitialWorkDirRequirement
    listing:
      - entryname: config_drops.ini
        entry: |
          [Drops]
          samtools = samtools
          star = STAR
          whitelistDir = /usr/app/baseqDrops/whitelist
          cellranger_ref_hg38 = $(inputs.index_dir.path)

inputs:
- id: index_dir
  type: Directory
```

The above tool produces a file called config_drops.ini in the working directory with 4 lines. The first three refer to paths in the docker image, the fourth line refers the input directory and will put the path generated by CWL into the config file.

10.3 Part 3: Making an input file or directory writable

If you need to make a file writable you can use the writable attribute:

```
requirements:
  - class: InitialWorkDirRequirement
    listing:
      - entry: $(inputs.input_file)
        writable: true

inputs:
- id: input_file
  type: File
```

CHAPTER 11

Cloud Providers

CHAPTER 12

Sharing Workflows

CHAPTER 13

Workflow Metadata

CHAPTER 14

CWL & Linked Data

Below is a somewhat verbose story/explanation of the connection between CWL and linked data.

- SALAD is a schema language for linked data, co-developed by developers of JSON-LD and Apache Avro
- the **Common Workflow Language (CWL)** is more of a standard/specification than a programming language or DSL, according to Michael Crusoe, Peter Amstutz, et al.
- **when you poke around the CWL GitHub repo long enough, you come across files like CommandLineTool.yml and Workfl**
  - when building tools and workflows, these are the primary "classes" of descriptor files (documents) that one would write, typically with a `.cwl` extension
  - as far as I can tell the `.yml` files represent the corresponding schema for each class, based on SALAD

The link between CWL and JSON-LD isn't obvious at first, because a .cwl descriptor is typically more YAML-like in structure; however, a few things happen when a CWL file is preprocessed before execution (you can test this with the `cwltool --print-pre my_tool-or-workflow.cwl` command):

- the output is clearly JSON, not YAML
- namespaces and references get expanded:

```
$namespaces:
dct: http://purl.org/dc/terms/
foaf: http://xmlns.com/foaf/0.1/

dct:creator:
"@id": "http://orcid.org/0000-0001-9758-0176"
foaf:name: James Eddy
foaf:mbox: "mailto:james.a.eddy@gmail.com"
```

becomes...

```
{
    "$namespaces": {
        "dct": "http://purl.org/dc/terms/",
```

(continues on next page)

(continued from previous page)

```
        "foaf": "http://xmlns.com/foaf/0.1/"
    },
    "http://purl.org/dc/terms/creator": {
        "@id": "http://orcid.org/0000-0001-9758-0176",
        "http://xmlns.com/foaf/0.1/mbox": "mailto:james.a.eddy@gmail.com",
        "http://xmlns.com/foaf/0.1/name": "James Eddy"
    },

    ...
```

paths get resolved (ish):

```
inputs:
template_file:
    type: File
    inputBinding:
    position: 1

input_file:
    type: File
    inputBinding:
    position: 2
```

becomes…

```
"inputs": [
    {
        "id": "file:///Users/jaeddy/code/github/containers/dockstore-workflow-
↪helloworld/dockstore-tool-helloworld.cwl#input_file",
        "inputBinding": {
            "position": 2
        },
        "type": "File"
    },
    {
        "id": "file:///Users/jaeddy/code/github/containers/dockstore-workflow-
↪helloworld/dockstore-tool-helloworld.cwl#template_file",
        "inputBinding": {
            "position": 1
        },
        "type": "File"
    }
],
```

at runtime, the "job" JSON (or YAML) file gets used and processed somehow to define the actual paths:

```
{
    "template_file": {
        "class": "File",
        "path": "template.txt"
    },
    "input_file": {
        "class": "File",
        "path": "input.txt"
    }
}
```

(result not shown — because I'm not entirely sure how to produce it)

So… there are clearly some JSON-LD "things" going on here. Further evidence as you scan through the schemas are sections like this:

```
- name: "class"
  jsonldPredicate:
    "_id": "@type"
    "_type": "@vocab"
  type: string
```

In terms of developing workflows for Translator using CWL, we could stick to a file-centric approach: dumping the results from one query/task into a JSON, then passing that as input to the next query/task — leaving it up to the underlying software to handle logic related to parsing and validation. However, I think we could take advantage of CWL's JSON-LD elements to operate directly on the data objects, and utilize schemas/namespaces/ontologies to specify and validate the more "conceptual" inputs and outputs (i.e., not just file formats). I've played around with this idea a bit using CWL's `SchemaDefRequirement` to schematize special input `record` types…

```
requirements:
- class: InlineJavascriptRequirement
- class: SchemaDefRequirement
    types:
    - $import: biolink-types.yaml
biolink-types.yaml

type: record
name: disease
fields:
- name: thing_id
type: string
- name: thing_name
type: string
- name: thing_category
type: string
```

Such that I can now parameterize the CWL tool with my `"disease"` record:

```
{
    "disease": {
        "thing_id": "8712",
        "thing_name": "neurofibromatosis",
        "thing_category": ""
    }
}
```

… and use some funny in-line JavaScript to parse and pass that record to the Python module as a JSON string:

```
inputs:
- id: disease
    label: Disease
    type: biolink-types.yaml#disease
    inputBinding:
    position: 1
    valueFrom: $(JSON.stringify(inputs.disease))
```

This would hopefully allow me to take advantage of a CWL executor's built-in features for validating parameters, such that I'd get an error or warning if my `"disease"` didn't conform to specs. It'd also be nice if we could verify that identifiers and other values map to allowable vocabularies, based on the BioLink model.

This is where I get a bit lost/stuck, and haven't quite been able to wrap my head around the details or mechanics…

CHAPTER 15

Synapse CWL Tools

CHAPTER 16

Synapse Workflow Hook

CHAPTER 17

Indices and tables

- genindex
- modindex
- search