How to create a CWL Workflow for AWE

workflow for execution with Skyport

Goal: Build a two-step workflow in CWL and submit CWL workunits to AWE for distributed computing of many wordclouds.

This example CWL workflow converts PDFs via pdftotext and uses wordcloud to create the cloud images.

Requirements:

- Skyport setup from day 1 including the repository
- Docker

Hint: Word Cloud install instructions can be found here: https://github.com/amueller/word_cloud

Starting with the PDF converter

1. Setup execution environment
2. Create CWL tool

Start the Skyport2 stack (awe server, shock server) described in Skyport2 Quick Start Guide

Navigate to the Skyport2 repository, run the following initialization command, and change to **CWL/wordcloud-tutorial**:

```
git clone http://github.com/MG-RAST/Skyport2
cd Skyport2
source ./init-client.sh
      # sets some environment variables to submit
      SKYPORT_HOST=172.18.0.1
      AWE_SERVER =http://172.18.0.1:8001/awe/api/
      SHOCK_SERVER=http://172.18.0.1:8001/shock/api/
cd CWL/wordcloud-tutorial
```

Create Docker container

1. `template.dockerfile` contains the boilerplate that we need to build a blank docker container running ubuntu. Here it is:

```
# Build from base image
FROM ubuntu:latest


# Update base images with latest patches
RUN apt-get update && apt-get upgrade -y


# Install dependencies
# RUN apt-get install -y \
#              ...



# Add additional commands
# RUN <COMMAND>

# Set entrypoint
# ENTRYPOINT [ "ls" ]
```

2. `demo2.dockerfile` has just this boilerplate plus one more line:

```
RUN apt-get install -y poppler-utils
```

`poppler-utils` is a collection of pdf-manipulating utilities.

```
# Build from base image
FROM ubuntu:latest


# Update base images with latest patches
RUN apt-get update && apt-get upgrade -y


# Install dependencies
RUN apt-get install -y poppler-utils
```

3. The syntax for building a container is:
   `docker build -t <name>:<tag> -f Dockerfile .`

Here we will build a container named demo:2
If you run into problems try the `--no-cache` option.

```
docker build  -t demo:2   -f demo2.dockerfile .
    Sending build context to Docker daemon  221.2MB
    Step 1/3 : FROM ubuntu:latest
…
    Successfully built 8e10b6276b07
```

4. Now that we’ve built demo:2 and it has the pdf-manipulation suite, we can run it in a container:
   `docker run --rm demo:2 pdftotext --help`

```
docker run --rm demo:2 pdftotext --help
    pdftotext version 0.41.0
    Copyright 2005-2016 The Poppler Developers - 
http://poppler.freedesktop.org
    Copyright 1996-2011 Glyph & Cog, LLC
    Usage: pdftotext [options] <PDF-file> [<text-file>]
     -f <int>             : first page to convert
     -l <int>             : last page to convert
     -r <fp>              : resolution, in DPI (default is 72)
     -x <int>             : x-coordinate of the crop area top 
left corner
     -y <int>             : y-coordinate of the crop area top 
left corner
     -W <int>             : width of crop area in pixels 
(default is 0)
     -H <int>             : height of crop area in pixels 
(default is 0)
     -layout              : maintain original physical layout
     -fixed <fp>          : assume fixed-pitch (or tabular) 
text
     -raw                 : keep strings in content stream 
order
```

```
  -q                   : don't print any messages or errors
  -v                   : print copyright and version info
  -h                   : print usage information
  -help                : print usage information
  --help               : print usage information
  -?                   : print usage information
```

Create CWL tool for pdftotext

1. `template.tool.cwl` contains the basic elements of the cwl syntax:

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.0

# Type of definition
#   CommandLineTool , Workflow , ExpressionTool

class:  CommandLineTool

# optional label
# label: <LABEL FOR CWL TOOL>
# optional description/documentation
# doc: <DETAILED DESCRIPTION>

# optional hints for CWL execution
# hints:
# set execution environment for baseCommand
#  - class: DockerRequirement
#    dockerPull: <DOCKER IMAGE NAME>


# required, name of command line tool
baseCommand: <COMMAND>

# optional
# arguments: <LIST OF CONSTANT OR DERIVED COMMAND LINE OPTIONS>

# required, input mapping
inputs: <LIST OF INPUT OPTIONS AND MAPPING TO COMMAND LINE>

# output mapping
outputs: <LIST OF NAMED OUTPUTS AND MAPPING TO COMMAND LINE
```

```
TOOL OUTPUT>
```

These define the inputs, the outputs, and the detailed command line syntax for our pdftotext command. In this case the cwl file names the docker container that has the pdftotext command installed. Here are some of the modifications in `pdftotext.cwl`

2. We set the label to PDF-to-Text:

```
# optional label
label: PDF-to-Text
```

3. We set the name of the executable:

```
# required, name of command line tool
baseCommand: pdftotext
```

4. The hints: dockerPull field describes the Docker image we need to run the command:

```
hints:
# set execution environment for baseCommand
- class: DockerRequirement
  dockerPull: demo:2
```

5. The script has two **positional** arguments, the PDF input filename and the text output filename:

```
inputs:
  pdf:
    type: File
    doc: PDF input file to extract text from
    inputBinding:
      position: 1
  text:
    type: string
    doc: Name for text output file
    inputBinding:
      position: 2
```

6. And finally we need to find the output file. Here we are giving the name extractedText to the output. We already have a variable, inputs.text, with the name of the output file.

```
# output mapping
outputs:
  extractedText:
    type: File
    outputBinding:
      glob: $(inputs.text)
```

7. All of this is put together in `pdftotext.cwl`:

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.0

# Type of definition
#   CommandLineTool , Workflow , ExpressionTool

class:  CommandLineTool

# optional label
label: PDF-to-Text
# optional description/documentation
doc: CWL wrapper for pdftotext

hints:
# set execution environment for baseCommand
- class: DockerRequirement
  dockerPull: demo:2

# required, name of command line tool
baseCommand: pdftotext

inputs:
  pdf:
    type: File
    doc: PDF input file to extract text from
    inputBinding:
      position: 1
  text:
    type: string
    doc: Name for text output file
    inputBinding:
      position: 2

# output mapping
outputs:
  extractedText:
    type: File
    outputBinding:
      glob: $(inputs.text)
```

8. If you have cwl-runner in your path, you can run pdftotext from cwl and display the command's help:

   ```
   cwl-runner pdftotext.cwl --help
   ```

   If you do not have cwl-runner installed, you can run it in a docker container. The mgrast/awe-submitter:develop container not-so-coincidentally has cwl-runner.

   ```
   docker run -v `pwd`/pdftotext.cwl:/pdftotext.cwl
   mgrast/awe-submitter:develop cwl-runner /pdftotext.cwl --help
   ```

```
/usr/bin/cwl-runner 1.0.20180116213856
Resolved '/pdftotext.cwl' to 'file:///pdftotext.cwl'
usage: /pdftotext.cwl [-h] --pdf PDF --text TEXT [job_order]

positional arguments:
  job_order    Job input json file

optional arguments:
  -h, --help   show this help message and exit
  --pdf PDF    PDF input file to extract text from
  --text TEXT  Name for text output file
```

9. Create a submission directory and and add a pdf file into it, e.g.:

```
mkdir -p submission
cp ../../CWL/Data/demo.pdf submission/
```

10. Create a CWL job file for pdftotext tool and save it as job.yaml in the parent directory of the submission directory/ The tool has two input parameter:
    a. `pdf` -> points to a file
    b. `text` -> name for output file

```
pdf:
  class: File
  path: submission/demo.pdf
text: demo.txt
```

11. Submit workflow and job:

```
# submit pdftotext.cwl  workflow with job.yaml to AWE
docker run -ti --network skyport2_default --rm \
-v `pwd`:/mnt \
mgrast/awe-submitter:develop \
/go/bin/awe-submitter   \
--pack   \
--wait   \
--shockurl=${SHOCK_SERVER}  \
--serverurl=${AWE_SERVER}  \
/mnt/pdftotext.cwl \
/mnt/job.yaml
```

```
{
    "extractedText": {
        "class": "File",
        "location":
"http://shock:7445/node/905863b6-bdde-40de-bd98-e7ff44903d51?download"
,
        "path":
"/Users/Andi/Development/MG-RAST-Repo/Skyport2/live-data/awe-worker/wo
rk/e7/8d/63/e78d63df-768c-4f3e-8f76-0c87221d00fe__entrypoint_wrapper_s
tep_0/tmp/nWI0JO/demo.txt",
        "basename": "demo.txt",
        "dirname": "",
        "nameroot": "",
        "nameext": "",
        "checksum": "sha1$b6272ceb6da71f9a1ed61069dfde46856851fb70",
        "size": 40649,
        "secondaryFiles": null,
        "format": "",
        "contents": ""
    }
}
```

12. To retrieve the workflow output open the Shock Browser (at http://http://localhost:8001/shock/) and download the files.

13. You can alternately get the results from the SHOCK API using a command like

```
curl \
"http://localhost:8001/shock/api/node/905863b6-bdde-40de-bd98-e7ff44903d51?download" \
-o demo.txt
```

WordCloud

Extend existing Dockerfile for wordcloud

1. To make wordclouds, we're going to need pip (to install the wordcloud packages) and we'll need to install the wordcloud package .

   So to `demo2.dockerfile` we will add the final two lines below:

```
# Build from base image
```

```
FROM ubuntu:latest

# Update base images with latest patches
RUN apt-get update && apt-get upgrade -y

# Install dependencies
RUN apt-get install -y \
  poppler-utils \
  python-pip
RUN pip install wordcloud
```

3. And we will rebuild the container with

```
docker build -t demo:2 -f wordCloud.dockerfile .
```

```
Sending build context to Docker daemon  4.243MB
Step 1/5 : FROM ubuntu:latest
 ---> 0ef2e08ed3fa
Step 2/5 : RUN apt-get update -y && apt-get upgrade -y
…
 ---> a2228adeef70
Successfully built a2228adeef70
Successfully tagged demo:2
```

4. Our container now has the tools for a two-step workflow, extracting text and producing a wordcloud from the text. We can see the CLI help by running the wordcloud tool thusly:

```
docker run -t demo:2 wordcloud_cli.py --help
```

Create CWL wordcloud tool

1. Wordcloud.cwl must be written around the command line specification that we saw running wordcloud_cli.py --help.

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.0

# Type of definition
#   CommandLineTool , Workflow , ExpressionTool

class:  CommandLineTool
```

```
# optional label
label: Word-Cloud
# optional description/documentation
doc: Makes images containing words of varying sizes

# optional hints for CWL execution
hints:
# set execution environment for baseCommand
  - class: DockerRequirement
    dockerPull: demo:2


# required, name of command line tool
baseCommand: wordcloud_cli.py

# optional
# arguments: <LIST OF CONSTANT OR DERIVED COMMAND LINE OPTIONS>

# required, input mapping
inputs:
  text:
    type: File
    doc: input file to create wordcloud image from
    inputBinding:
      prefix: --text
  outname:
    type: string
    doc: Name for text output file
    inputBinding:
      prefix: --imagefile

# output mapping
outputs:
  image:
    type: File
    outputBinding:
      glob: $(inputs.outname)
```

And here is a discussion of the parts:

14. The executable name:

```
baseCommand: wordcloud_cli.py
```

15. We have extended the Dockerfile from the pdftotext example and rebuild the docker image for the command line tools, so we will use the same image name as before:

```
hints:
# set execution environment for baseCommand
- class: DockerRequirement
# dockerPull: <NAME>:<TAG>
  dockerPull: demo:2
```

16. The script has to main arguments, text input file and image output name file:

```
usage: wordcloud_cli.py [-h] [--text file] [--stopwords file]
                        [--imagefile file] [--fontfile path] [--mask file]
                        [--colormask file] [--relative_scaling rs]
                        [--margin width] [--width width] [--height height]
                        [--color color] [--background color]
                        [--no_collocations]
```

Adding two **named** input options *`text`* and *`outname`* to the **`inputs`** section. *`text`* is a file and *`outname`* a string.

```
inputs:
  text:
    type: File
    doc: input file to create wordcloud image from
    inputBinding:
      prefix: --text
  outname:
    type: string
    doc: Name for text output file
    inputBinding:
      prefix: --imagefile
```

17. Collect the output, the result is an image file with the value from *`outname`* as name:

```
# output mapping
outputs:
  image:
    type: File
    outputBinding:
      glob: $(inputs.outname)
```

18. Test CWL tool, if you have cwl-runner in your path:

```
cwl-runner wordcloud.cwl --help
```

otherwise:

```
docker run -v `pwd`/wordcloud.cwl:/wordcloud.cwl
mgrast/awe-submitter:develop cwl-runner /wordcloud.cwl --help
```

```
usage: /wordcloud.cwl [-h] --outname OUTNAME --text TEXT
[job_order]

positional arguments:
  job_order           Job input json file

optional arguments:
  -h, --help          show this help message and exit
  --outname OUTNAME   Name for text output file
  --text TEXT         input file to create wordcloud image from
```

Combine tools into workflow

The Skyort2 repository is checked out. Change into the Skyport2 directory. We will create a workflow which takes a PDF and creates an image file with same prefix as PDF file.

Creating first workflow with pdftotext tool

1. CWL/Workflows/template.workflow.cwl summarizes the CWL workflow syntax in metasyntactic variables:

```
cwlVersion: v1.0
class: Workflow

# optional - additional requirements to execute this workflow
# requirements:
#   - class: InlineJavascriptRequirement

# required, workflow input mapping
inputs: <LIST OF INPUT OPTIONS AND MAPPING TO TOOLs/STEPs LINE>


# output mapping
outputs: <LIST OF NAMED OUTPUTS AND MAPPING \
          FROM TOOL OUTPUT TO WORKFLOW OUTPUT>


steps: <LIST OF WORKFLOW STEPS>

  <STEP NAME>:
    label: <TEXT>
    doc:  <TEXT>
    # required - cwl tool description
    run: <PATH TO CWL TOOL FILE>

    # required - step inputs
    in: <MAPPING OF TOOL INPUT OPTIONS TO \
           WORKFLOW INPUT OR STEP OUTPUT>

    # required , step outputs
    out: <LIST OF TOOL OUTPUTs>
```

This allows the definition of multiple steps.

2. `pdf2wordcloud1.cwl` has been adapted to describe the first step of this workflow:

```
cwlVersion: v1.0
class: Workflow

# optional - additional requirements to execute this workflow
 requirements:
  - class: StepInputExpressionRequirement

# required, workflow input mapping
inputs:
  pdf:
    type: File
    doc: PDF file for text extraction


# output mapping
# outputs: <LIST OF NAMED OUTPUTS AND MAPPING \
#          FROM TOOL OUTPUT TO WORKFLOW OUTPUT>


steps:
  # step name
  pdf2text:
    label: pdf2text
    doc: extract ascii text from PDF
    # path to tool
    run: ../Tools/pdftotext.cwl
    # assign values to step/tool inputs
    in:
      # assign workflow input to tool input:
      # <tool input name>:<workflow input name>
      pdf: pdf
      text:
        # assign constant text
        default: "just-words.txt"

    out: [extractedText]

# mapping of output parameter to step outputs
outputs:
  # name of output parameter
  words:
    type: File
    # assign value from specified step output to output
parameter
    outputSource: pdf2text/extractedText
```

3. This workflow has a single input parameter, a PDF file. Inputs are defined similar to tool inputs but without specified `inputBinding`:

```
inputs:
  pdf:
    type: File
    doc: PDF file for text extraction
```

4. The first step extracts ascii text from a PDF file, for this we will use the pdftotext tool:

```
# list of workflow steps
steps:
  # step name
  pdf2text:
    label: pdf2text
    doc: extract ascii text from PDF
    # path to tool
    run: ../Tools/pdftotext.cwl
    in: ...

```

5. The tool requires two input parameter `pdf` and `text`. First we will map the workflow input for the PDF file to the corresponding tool input, second we will set a value for the output filename required by the tool:

```
    # assign values to step/tool inputs
    in:
      # assign workflow input to tool input:
      # <tool input name>:<workflow input name>
      pdf: pdf
      text:
        # assign constant text
        default: "just-words.txt"
```

6. The tool creates an output named `extractedText` and the step has to capture the output for further processing:

```
# list of tool outputs to be captured for downstream processing
    out: [extractedText]
```

7. To finish the workflow and test the first step the workflow `outputs` section is required. For this we will return the extracted ascii text:

```
# mapping of output parameter to step outputs
outputs:
  # name of output parameter
  words:
    type: File
    # assign value from specified step output to output
parameter
    outputSource: pdf2text/extractedText
```

8. Assuming we have following directory structure:
    a. ./Workflows for CWL workflows
    b. ./Tools for CWL tools
    c. ./Data for job files and workflow/tool input data

   Create a job file in ./Data and add your PDF to it (e.g. pdf2wordcloud.job.yaml and demo.pdf):

```
pdf:
  class: File
  path: demo.pdf
```

Then execute a submitter, e.g.:

```
#  submit pdf2wordcloud1.cwl  workflow with pdf2wordcloud.job.yaml to AWE
docker run -ti --network skyport2_default --rm \
-v `pwd`:/mnt \
mgrast/awe-submitter:develop \
/go/bin/awe-submitter   \
--pack   \
--wait \
--shockurl=${SHOCK_SERVER}  \
 --serverurl=${AWE_SERVER}  \
/mnt/pdf2wordcloud1.cwl \
/mnt/pdf2wordcloud.job.yaml
```

When this job completes, we get a json-formatted reciept, and the output data is viewable in the Shock Browser.

Adding additional step to workflow

9. `pdf2wordcloud.cwl` has a second workflow step `text2wordCloud`. It takes the output from `pdf2text`, runs `text2wordCloud` on the result, and stores the final result (an image) in SHOCK. This `text2wordCloud` step will run the `wordcloud.cwl` tool :

```
  # second step
  text2wordCloud:
    label: word-cloud
    doc: create png from text file
    # path to tool
    run: ../Tools/wordcloud.cwl
    # assign values to step/tool inputs
    in:
      # assign output from previous step to tool input:
      # <tool input name>:<previous step/tool output name>
      text: pdf2text/extractedText
      outname:
        # derive output name from pdf input filename
        source: pdf
        valueFrom: $(self.nameroot).png
    # return output from tool
    out: [image]
```

10. Notice we are using the `source` and `valueFrom` fields. The part within **`$(...)`** is treated as an expression. For this to work add `StepInputExpressionRequirement` to the workflow `requirements`:

```
requirements:
 - class: StepInputExpressionRequirement
```

11. Last capture the output from `text2wordCloud` step:

```
outputs:
  # name of output parameter
  words:
    type: File
    outputSource: pdf2text/extractedText
  wordCloudImage:
    type: File
    outputSource: text2wordCloud/image
```

12. EXERCISE: Try and submit the complete workflow which will produce extracted.txt.png.