Popper Documentation

Release 2.x

Ivo Jimenez

Nov 05, 2020

Contents

CHAPTER 1

Getting Started

Before going through this guide, you need to have the Docker engine installed on your machine (see installations instructions here). In addition, this guide assumes familiarity with Linux containers and the container-native paradigm to software development. You can read a high-level introduction to these concepts in this page, where you can also find references to external resources that explain them in depth.

1.1 Installation

To install or upgrade Popper, run the following in your terminal:

```
curl -sSfL https://raw.githubusercontent.com/getpopper/popper/master/install.sh | sh
```

1.2 Create Your First Workflow

Assume that as part of our work we want to carryout two tasks:

1. Download a dataset (CSV) that we know is available at https://github.com/datasets/co2-fossil-global/raw/master/global.csv
2. Modify the dataset, specifically we want to get the transpose of the this CSV table.

For the first task we can use `curl`, while for the second we can use `csvtool`.

When we work under the container-native paradigm, instead of going ahead and installing these on our computer, we first look for available images on a container registry, for example https://hub.docker.com, to see if the software we need is available.

In this case we find two images that do what we need and proceed to write this workflow in a `wf.yml` file using your favorite editor:

```
steps:
# download CSV file with data on global CO2 emissions
- id: download
  uses: docker://byrnedo/alpine-curl:0.1.8
  args: [-LO, https://github.com/datasets/co2-fossil-global/raw/master/global.csv]

# obtain the transpose of the global CO2 emissions table
- id: get-transpose
  uses: docker://getpopper/csvtool:2.4
  args: [transpose, global.csv, -o, global_transposed.csv]
```

1.3 Run your workflow

To execute the workflow you just created:

```
popper run -f wf.yml
```

Since this workflow consists of two steps, there were two corresponding containers that were executed by the underlying container engine, which is Docker in this case. We can verify this by asking Docker to show the list of existing containers:

```
docker ps -a
```

You should see the two containers from the example workflow being listed along with other containers. The name of the containers created by popper are prefixed with `popper_`. To obtain more detailed information of what the `popper run` command does, you can pass the `--help` flag to it:

```
popper run --help
```

> **TIP**: All popper subcommands allow you to pass `--help` flag to it to get more information about what the command does.

1.4 Debug your workflow

From time to time, we find ourselves with a step that does not quite do what we want it to. In these cases, we can open an interactive shell instead of having to update the YAML file and invoke `popper run` again. In those cases, the `popper sh` comes handy. For example, if we would like to explore what other things can be done inside the container for the second step:

```
popper sh -f wf.yml get-transpose
```

And the above opens a shell inside a container instantiated from the `docker.io/getpopper/csvtool:2.4` image. In this shell we can, for example, obtain information about what else can the `csvtool` do:

```
csvtool --help
```

Based on this exploration, we can see that we can pass a `-u TAB` flag to the `csvtool` in order to generate a tab-separated output file instead of a comma-separated one. Assuming this is what we wanted to achieve in our case, we then quit the container by running `exit`.

Back on our host machine context, that is, not running inside the container anymore, we can update the second step by editing the YAML file to look like the following:

```
- id: get-transpose
  uses: docker://getpopper/csvtool:2.4
  args: [transpose, global.csv, -u, TAB, -o, global_transposed.csv]
```

And test that what we changed worked by running in non-interactive mode again:

```
popper run -f wf.yml get-transpose
```

1.5 Next Steps

- Learn more about all the CLI features that Popper provides.
- Take a look at the "Workflow Language" for the details on what else can you specify as part of a Step's attributes.
- Read the "Popper Execution Runtime" section to learn more about what other execution environments Popper supports, as well as how to customize the behavior of the underlying execution.
- Browse existing workflow examples.
- Take a self-paced tutorial to learn how to use other features of Popper.

CHAPTER 2

CLI feautures

2.1 New workflow initialization

Create a Git repository:

```
mkdir mypaper
cd mypaper
git init
echo '# mypaper' > README.md
git add .
git commit -m 'first commit'
```

Initialize the popper repository and add the configuration file to git:

```
popper init
git add .
git commit -m 'adds .popper.yml file'
```

Initialize a workflow

```
popper scaffold
```

Show what this did (a `wf.yml` should have been created):

```
ls -l
```

Commit the "empty" pipeline:

```
git add .
git commit -m 'adding my first workflow'
```

2.2 Executing a workflow

To run the workflow:

```
popper run -f wf.yml
```

where `wf.yml` is a file containing a workflow.

2.3 Executing a step interactively

For debugging a workflow, it is sometimes useful to open a shell inside a container associated to a step of a workflow. To accomplish this, run:

```
popper sh <STEP>
```

where `<STEP>` is the name of a step contained in the workflow. For example, given the following workflow:

```
steps:
- id: mystep
  uses: docker://ubuntu:18.04
  runs: ["ls", "-l"]
  dir: /tmp/
  env:
    MYENVVAR: "foo"
```

if we want to open a shell that puts us inside the `mystep` above (inside an container instance of the `ubuntu:18.04` image), we run:

```
popper sh mystep
```

And this opens an interactive shell inside that step, where the environment variable `MYENVVAR` is available. Note that the `runs` and `args` attributes are overridden by Popper. By default, `/bin/bash` is used to start the shell, but this can be modified with the `--entrypoint` flag.

2.4 Customizing container engine behavior

By default, Popper instantiates containers in the underlying engine by using basic configuration options. When these options are not suitable to your needs, you can modify or extend them by providing engine-specific options. These options allow you to specify fine-grained capabilities, bind-mounting additional folders, etc. In order to do this, you can provide a configuration file to modify the underlying container engine configuration used to spawn containers. This is a YAML file that defines an `engine` dictionary with custom options and is passed to the `popper run` command via the `--conf` (or `-c`) flag.

For example, to make Popper spawn Docker containers in privileged mode, we can write the following option:

```
engine:
    name: docker
    options:
       privileged: True
```

Similarly, to bind-mount additional folders, we can use the `volumes` option to list the directories to mount:

```
engine:
    name: docker
    options:
       privileged: True
       volumes:
       - myvol1:/folder
       - myvol2:/app
```

Assuming the above is stored in a file called `config.yml`, we pass it to Popper by running:

```
popper run -f wf.yml -c config.yml
```

NOTE:

Currently, the `--conf` option is only supported for the `docker`engine.

2.5 Continuously validating a workflow

The `ci` subcommand generates configuration files for multiple CI systems. The syntax of this command is the following:

```
popper ci --file wf.yml <service-name>
```

Where `<name>` is the name of CI system (see `popper ci --help` to get a list of supported systems). In the following, we show how to link github with some of the supported CI systems. In order to do so, we first need to create a repository on github and upload our commits:

```
# set the new remote
git remote add origin <your-github-repo-url>

# verify the remote URL
git remote -v

# push changes in your local repository up to github
git push -u origin master
```

2.5.1 TravisCI

For this, we need an account at Travis CI. Assuming our Popperized repository is already on GitHub, we can enable it on TravisCI so that it is continuously validated (see here for a guide). Once the project is registered on Travis, we proceed to generate a `.travis.yml` file:

```
cd my-popper-repo/
popper ci --file wf.yml travis
```

And commit the file:

```
git add .travis.yml
git commit -m 'Adds TravisCI config file'
```

We then can trigger an execution by pushing to GitHub:

```
git push
```

After this, one go to the TravisCI website to see your pipelines being executed. Every new change committed to a public repository will trigger an execution of your pipelines. To avoid triggering an execution for a commit, include a line with `[skip ci]` as part of the commit message.

NOTE: TravisCI has a limit of 2 hours, after which the test is terminated and failed.

2.5.2 CircleCI

For CircleCI, the procedure is similar to what we do for TravisCI (see above):

1. Sign in to CircleCI using your github account and enable your repository.
2. Generate config files and add them to the repo:

```
cd my-popper-repo/
popper ci --file wf.yml circle
git add .circleci
git commit -m 'Adds CircleCI config files'
git push
```

2.5.3 GitLab-CI

For GitLab-CI, the procedure is similar to what we do for TravisCI and CircleCI (see above), i.e. generate config files and add them to the repo:

```
cd my-popper-repo/
popper ci --file wf.yml gitlab
git add .gitlab-ci.yml
git commit -m 'Adds GitLab-CI config file'
git push
```

If CI is enabled on your instance of GitLab, the above should trigger an execution of the pipelines in your repository.

2.5.4 Jenkins

For Jenkins, generating a `Jenkinsfile` is done in a similar way:

```
cd my-popper-repo/
popper ci --file wf.yml jenkins
git add Jenkinsfile
git commit -m 'Adds Jenkinsfile'
git push
```

Jenkins is a self-hosted service and needs to be properly configured in order to be able to read a github project with a `Jenkinsfile` in it. The easiest way to add a new project is to use the Blue Ocean UI. A step-by-step guide on how to create a new project using the Blue Ocean UI can be found here. In particular, the `New Pipeline from a Single Repository` has to be selected (as opposed to `Auto-discover Pipelines`).

2.6 Visualizing workflows

While `.workflow` files are relatively simple to read, it is nice to have a way of quickly visualizing the steps contained in a workflow. Popper provides the option of generating a graph for a workflow. To generate a graph for a pipeline, execute the following:

```
popper dot -f wf.yml
```

The above generates a graph in `.dot` format. To visualize it, you can install the `graphviz` package and execute:

```
popper dot -f wf.yml | dot -T png -o wf.png
```

The above generates a `wf.png` file depicting the workflow. Alternatively you can use the http://www.webgraphviz.com/ website to generate a graph by copy-pasting the output of the `popper dot` command.

CHAPTER 3

Concepts

The main three concepts behind Popper are Linux containers, the container-native paradigm, and workflows. This page is under construction, we plan on expanding it with our own content (contributions are more than welcome)! For now, we provide with a list of external resources and a Glossary.

3.1 Resources

Container Concepts:

- Overview of Containers in Red Hat Systems (Red Hat)
- An Introduction to Containers (Rancher)
- A Beginner-Friendly Introduction to Containers, VMs and Docker (freecodecamp.org)
- A Practical Introduction to Container Terminology (Red Hat)

Container-native paradigm:

- 5 Reasons You Should Be Doing Container-native Development (Microsoft)
- Let's Define "Container-native" (TechCrunch)
- The 7 Characteristics of Container-native Infrastructure (Joyent)

Docker:

- A Docker tutorial for beginners
- Dockerfile tutorial by example

Singularity:

- Introduction to Singularity

3.2 Glossary

- **Linux containers**. An OS-level virtualization technology for isolating applications in a Linux host machine.
- **Container runtime**. The software that interacts with the Linux kernel in order to provide with container primitives to upper-level components such as a container engine (see "Container Engine"). Examples of runtimes are runc, Kata and crun.
- **Container engine**. Container management software that provides users with an interface to. Examples of engines are Docker, Podman and Singularity.
- **Container-native development**. An approach to writing software that makes use of containers at every stage of the software delivery cycle (building, testing, deploying, etc.). In practical terms, when following a container-native paradigm, other than a text editor or ID, dependencies required to develop, test or deploy software are NEVER installed directly on your host computer. Instead, they are packaged in container images and you make use of them through a container engine.
- **Workflow**. A series of steps, where each step specifies what it does, as well as which other steps need to be executed prior to its execution. It is commonly represented as a directed acyclic graph (DAG), where each node represents a step. The word "pipeline" is usually used interchangeably to refer to a workflow.
- **Task or Step**. A node in a workflow DAG.
- **Container-native workflow**. A workflow where each step runs in a container.
- **Container-native task or step**. A step in a container-native workflow that specifies the image it runs, the arguments that are executed, the environment available inside the container, among other attributes available for containers (network configuration, resource limits, capabilities, volumes, etc.).

CHAPTER 4

Workflow Syntax and Execution Runtime

This section introduces the YAML syntax used by Popper, describes the workflow execution runtime and shows how to execute workflows in alternative container engines.

4.1 Syntax

A Popper workflow file looks like the following:

```
steps:
- uses: docker://alpine:3.9
  args: ["ls", "-la"]

- uses: docker://alpine:3.11
  args: ["echo", "second step"]

options:
  env:
    FOO: BAR
  secrets:
  - TOP_SECRET
```

A workflow specification contains one or more steps in the form of a YAML list named `steps`. Each item in the list is a dictionary containing at least a `uses` attribute, which determines the docker image being used for that step. An `options` dictionary specifies options that are applied to the workflow.

4.1.1 Workflow steps

The following table describes the attributes that can be used for a step. All attributes are optional with the exception of the `uses` attribute.

4.1.2 Referencing images in a step

A step in a workflow can reference a container image defined in a `Dockerfile` that is part of the same repository where the workflow file resides. In addition, it can also reference a `Dockerfile` contained in public Git repository. A third option is to directly reference an image published a in a container registry such as DockerHub. Here are some examples of how you can refer to an image on a public Git repository or Docker container registry:

It's strongly recommended to include the version of the image you are using by specifying a SHA or Docker tag. If you don't specify a version and the image owner publishes an update, it may break your workflows or have unexpected behavior.

In general, any Docker image can be used in a Popper workflow, but keep in mind the following:

- When the `runs` attribute for a step is used, the `ENTRYPOINT` of the image is overridden.
- The `WORKDIR` is overridden and `/workspace` is used instead (see *The Workspace* section below).
- The `ARG` instruction is not supported, thus building an image from a `Dockerfile` (public or local) only uses its default value.
- While it is possible to run containers that specify `USER` other than root, doing so might cause unexpected behavior.

4.1.3 Referencing private Github repositories

You can reference Dockerfiles located in private Github repositories by defining a `GITHUB_API_TOKEN` environment variable that the `popper run` command reads and uses to clone private repositories. The repository referenced in the `uses` attribute is assumed to be private and, to access it, an API token from Github is needed (see instructions here). The token needs to have permissions to read the private repository in question. To run a workflow that references private repositories:

```
export GITHUB_API_TOKEN=access_token_here
popper run -f wf.yml
```

If the access token doesn't have permissions to access private repositories, the `popper run` command will fail.

4.1.4 Workflow options

The `options` attribute can be used to specify `env` and `secrets` that are available to all the steps in the workflow. For example:

```
options:
  env:
    FOO: var1
    BAR: var2
  secrets: [SECRET1, SECRET2]

steps:
- uses: docker://alpine:3.11
  runs: sh
  args: ["-c", "echo $FOO $SECRET1"]

- uses: docker://alpine:3.11
  runs: sh
  args: ["-c", "echo $ONLY_FOR"]
  env:
    ONLY_FOR: this step
```

The above shows environment variables that are available to all steps that get defined in the `options` dictionary; it also shows an example of a variable that is available only to a single step (second step). This attribute is optional.

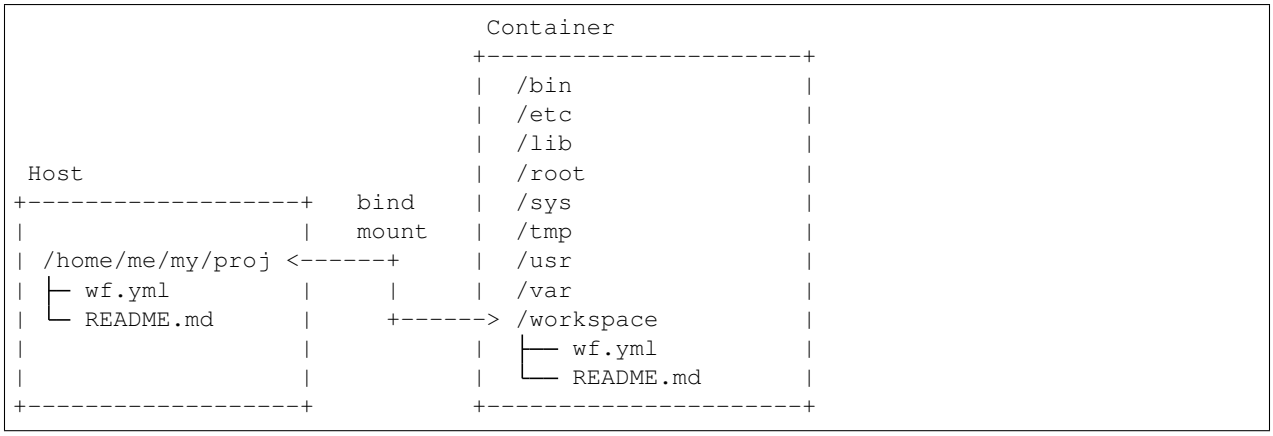
4.2 Execution Runtime

This section describes the runtime environment where a workflow executes.

4.2.1 The Workspace

When a step is executed, a folder in your machine is bind-mounted (shared) to the `/workspace` folder inside the associated container. By default, the folder being bind-mounted is `$PWD`, that is, the working directory from where `popper run` is being invoked from. If the `-w` (or `--workspace`) flag is given, then the value for this flag is used instead. See the official Docker documentation for more information about how volumes work with containers.

The following diagram illustrates this relationship between the filesystem namespace of the host (the machine where `popper run` is executing) and the filesystem namespace within container:



For example, let's look at a workflow that creates files in the workspace:

```
steps:
- uses: docker://alpine:3.12
  args: [touch, ./myfile]
```

The above workflow has only one single step that creates the `myfile` file in the workspace directory if it doesn't exist, or updates its metadata if it already exists, using the `touch` command. Assuming the above workflow is stored in a `wf.yml` file in `/home/me/my/proj/`, we can run it by first changing the current working directory to this folder:

```
cd /home/me/my/proj/
popper run -f wf.yml
```

And this will result in having a new file in `/home/me/my/proj/myfile`. However, if we invoke the workflow from a different folder, the folder being bind-mounted inside the container is a different one. For example:

```
cd /home/me/
popper run -f /home/me/my/proj/wf.yml
```

In the above, the file will be written to `/home/me/myfile`, because we are invoking the command from `/home/me/`, and this path is treated as the workspace folder. If we provide a value for the `--workspace` flag (or its short version `-w`), the workspace path then changes and thus the file is written to this given location. For example:

```
cd /
popper run -f /home/me/my/proj/wf.yml -w /home/me/my/proj/
```

The above writes the `/home/me/my/proj/myfile` even though Popper is being invoked from `/`. Note that the above is equivalent to the first example of this subsection, where we first changed the directory to `/home/me/my/proj` and ran `popper run -f wf.yml`.

4.2.2 Changing the working directory

To specify a working directory for a step, you can use the `dir` attribute in the workflow, which takes as value a string representing an absolute path inside the container. This changes where the specified command is executed. For example, adding `dir` as follows:

```
steps:
- uses: docker://alpine:3.9
  args: [touch, ./myfile]
  dir: /tmp/
```

And assuming that it is stored in `/home/me/my/proj/wf.yml`, invoking the workflow as:

```
cd /home/me
popper run -f wf.yml -w /home/me/my/proj
```

Would result in writing `myfile` in the `/tmp` folder that is **inside** the container filesystem namespace, as opposed to writing it to `/home/me/my/projc/` (the value given for the `--workspace` flag). As it is evident in this example, if the directory specified in the `dir` attribute resides outside the `/workspace` folder, then anything that gets written to it won't persist after the step ends its execution (see "Filesystem namespaces and persistence" below for more).

For completeness, we show an example of using `dir` to specify a folder within the workspace:

```
steps:
- uses: docker://alpine:3.9
  args: [touch, ./myfile]
  dir: /workspace/my/proj/
```

And executing:

```
cd /home/me
popper run -f wf.yml
```

would result in having a file in `/home/me/my/proj/myfile`.

4.2.3 Filesystem namespaces and persistence

As mentioned previously, for every step Popper bind-mounts (shares) a folder from the host (the workspace) into the `/workspace` folder in the container. Anything written to this folder persists. Conversely, anything that is NOT written in this folder will not persist after the workflow finishes, and the associated containers get destroyed.

4.2.4 Environment variables

A step can define, read, and modify environment variables. A step defines environment variables using the `env` attribute. For example, you could set the variables `FIRST`, `MIDDLE`, and `LAST` using this:

```
steps:
- uses: "docker://alpine:3.9"
  args: ["sh", "-c", "echo my name is: $FIRST $MIDDLE $LAST"]
  env:
    FIRST: "Jane"
    MIDDLE: "Charlotte"
    LAST: "Doe"
```

When the above step executes, Popper makes these variables available to the container and thus the above prints to the terminal:

```
my name is: Jane Charlotte Doe
```

Note that these variables are only visible to the step defining them and any modifications made by the code executed within the step are not persisted between steps (i.e. other steps do not see these modifications).

Git Variables

When Popper executes insides a git repository, it obtains information related to Git. These variables are prefixed with `GIT_` (e.g. to `GIT_COMMIT` or `GIT_BRANCH`).

4.2.5 Exit codes and statuses

Exit codes are used to communicate about a step's status. Popper uses the exit code to set the workflow execution status, which can be `success`, `neutral`, or `failure`:

4.3 Container Engines

By default, Popper workflows run in Docker on the machine where `popper run` is being executed (i.e. the host machine). This section describes how to execute in other container engines. See *next section* for information on how to run workflows on resource managers such as SLURM and Kubernetes.

To run workflows on other container engines, an `--engine <engine>` flag for the `popper run` command can be given, where `<engine>` is one of the supported ones. When no value for this flag is given, Popper executes workflows in Docker. Below we briefly describe each container engine supported, and lastly describe how to pass engine-specific configuration options via the `--conf` flag.

4.3.1 Docker

Docker is the default engine used by the `popper run`. All the container configuration for the docker engine is supported by Popper.

4.3.2 Singularity

Popper can execute a workflow in systems where Singularity 3.2+ is available. To execute a workflow in Singularity containers:

```
popper run --engine singularity
```

Limitations

- The use of `ARG` in `Dockerfiles` is not supported by Singularity.
- The `--reuse` flag of the `popper run` command is not supported.

4.3.3 Host

There are situations where a container runtime is not available and cannot be installed. In these cases, a step can be executed directly on the host, that is, on the same environment where the `popper` command is running. This is done by making use of the special `sh` value for the `uses` attribute. This value instructs Popper to execute the command or script given in the `runs` attribute. For example:

```
steps:
- uses: "sh"
  runs: ["ls", "-la"]

- uses: "sh"
  runs: "./path/to/my/script.sh"
  args: ["some", "args", "to", "the", "script"]
```

In the first step above, the `ls -la` command is executed on the workspace folder (see *"The Workspace"* section). The second one shows how to execute a script. Note that the command or script specified in the `runs` attribute are NOT executed in a shell. If you need a shell, you have to explicitly invoke one, for example:

```
steps:
- uses: sh
  runs: [bash, -c, 'sleep 10 && true && exit 0']
```

The obvious downside of running a step on the host is that, depending on the command being executed, the workflow might not be portable.

4.3.4 Custom engine configuration

Other than bind-mounting the `/workspace` folder, Popper runs containers with any default configuration provided by the underlying engine. However, a `--conf` flag is provided by the `popper run` command to specify custom options for the underlying engine in question (see here for more).

4.4 Resource Managers

Popper can execute steps in a workflow through other resource managers like SLURM besides the host machine. The resource manager can be specified either through the `--resource-manager/-r` option or through the config file. If neither of them are provided, the steps are run in the host machine by default.

4.4.1 SLURM

Popper workflows can run on HPC (Multi-Node environments) using Slurm as the underlying resource manager to distribute the execution of a step to several nodes. You can get started with running Popper workflows through Slurm by following the example below.

Let's consider a workflow `sample.yml` like the one shown below.

```
steps:
- id: one
  uses: docker://alpine:3.9
  args: ["echo", "hello-world"]

- id: two
  uses: popperized/bin/sh@master
  args: ["ls", "-l"]
```

To run all the steps of the workflow through slurm resource manager, use the `--resource-manager` or `-r` option of the `popper run` subcommand to specify the resource manager.

```
popper run -f sample.yml -r slurm
```

To have more finer control on which steps to run through slurm resource manager, the specifications can be provided through the config file as shown below.

We create a config file called `config.yml` with the following contents.

```
engine:
  name: docker
  options:
    privileged: True
    hostname: example.local

resource_manager:
  name: slurm
  options:
    two:
      nodes: 2
```

Now, we execute `popper run` with this config file as follows:

```
popper run -f sample.yml -c config.yml
```

This runs the step `one` locally in the host and step `two` through slurm on 2 nodes.

Host

Popper executes the workflows by default using the `host` machine as the resource manager. So, when no resource manager is provided like the example below, the workflow runs on the local machine.

```
popper run -f sample.yml
```

The above assumes `docker` as the container engine and `host` as the resource manager to be used.

CHAPTER 5

Guides

This is a list of guides related to several aspects of working with Popper workflows.

5.1 Choosing a location for your step

If you are developing a docker image for other people to use, we recommend keeping this image in its own repository instead of bundling it with your repository-specific logic. This allows you to version, track, and release this image just like any other software. Storing a docker image in its own repository makes it easier for others to discover, narrows the scope of the code base for developers fixing issues and extending the image, and decouples the image's versioning from the versioning of other application code.

5.2 Using shell scripts to define step logic

Shell scripts are a great way to write the code in steps. If you can write a step in under 100 lines of code and it doesn't require complex or multi-line command arguments, a shell script is a great tool for the job. When defining steps using a shell script, follow these guidelines:

- Use a POSIX-standard shell when possible. Use the `#!/bin/sh` shebang to use the system's default shell. By default, Ubuntu and Debian use the dash shell, and Alpine uses the ash shell. Using the default shell requires you to avoid using bash or shell-specific features in your script.
- Use `set -eu` in your shell script to avoid continuing when errors or undefined variables are present.

5.3 Hello world step example

You can create a new step by adding a `Dockerfile` to the directory in your repository that contains your step code. This example creates a simple step that writes arguments to standard output (`stdout`). An step declared in a `main.workflow` would pass the arguments that this step writes to `stdout`. To learn more about the instructions used in

the `Dockerfile`, check out the official Docker documentation. The two files you need to create an step are shown below:

./step/Dockerfile

```
FROM debian:9.5-slim

ADD entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

./step/entrypoint.sh

```
#!/bin/sh -l

sh -c "echo $*"
```

Your code must be executable. Make sure the `entrypoint.sh` file has `execute` permissions before using it in a workflow. You can modify the permission from your terminal using this command:

```
chmod +x entrypoint.sh
```

This `echos` the arguments you pass the step. For example, if you were to pass the arguments `"Hello World"`, you'd see this output in the command shell:

```
Hello World
```

5.4 Creating a Docker container

Check out the official Docker documentation.

5.5 Implementing a workflow for an existing set of scripts

This guide exemplifies how to define a Popper workflow for an existing set of scripts. Assume we have a project in a `myproject/` folder and a list of scripts within the `myproject/scripts/` folder, as shown below:

```
cd myproject/
ls -l scripts/

total 16
-rwxrwx---  1 user  staff   927B Jul 22 19:01 download-data.sh
-rwxrwx---  1 user  staff   827B Jul 22 19:01 get_mean_by_group.py
-rwxrwx---  1 user  staff   415B Jul 22 19:01 validate_output.py
```

A straight-forward workflow for wrapping the above is the following:

```
- uses: docker://alpine:3.12
  runs: "/bin/bash"
  args: ["scripts/download-data.sh"]

- uses: docker://alpine:3.12
  args: ["./scripts/get_mean_by_group.py", "5"]

- uses: docker://alpine:3.12
```

(continues on next page)

(continued from previous page)

```
  args [
    "./scripts/validate_output.py",
    "./data/global_per_capita_mean.csv"
  ]
```

The above runs every script within a Docker container. As you would expect, this workflow fails to run since the `alpine:3/12` image is a lightweight one (contains only Bash utilities), and the dependencies that the scripts need are not be available in this image. In cases like this, we need to either use an existing docker image that has all the dependencies we need, or create a docker image ourselves.

In this particular example, these scripts depend on CURL and Python. Thankfully, docker images for these already exist, so we can make use of them as follows:

```
- uses: docker://byrnedo/alpine-curl:0.1.8
  args: ["scripts/download-data.sh"]

- uses: docker://python:3.7
  args: ["./scripts/get_mean_by_group.py", "5"]

- uses: docker://python:3.7
  args [
    "./scripts/validate_output.py",
    "./data/global_per_capita_mean.csv"
  ]
```

The above workflow runs correctly anywhere where Docker containers can run.

CHAPTER 6

Other Resources

- A list of example workflows can be found at https://github.com/popperized/popper-examples.
- Self-paced hands-on tutorial.

CHAPTER 7

FAQ

7.1 How can I create a virtual environment to install Popper

The following creates a virtual environment in a `$HOME/venvs/popper` folder:

```
# create virtualenv
virtualenv $HOME/venvs/popper

# activate it
source $HOME/venvs/popper/bin/activate

# install Popper in it
pip install popper
```

The first step is is only done once. After closing your shell, or opening another tab of your terminal emulator, you'll have to reload the environment (`activate it` line above). For more on virtual environments, see here.

7.2 How can we deal with large datasets? For example I have to work on large data of hundreds GB, how would this be integrated into Popper?

For datasets that are large enough that they cannot be managed by Git, solutions such as a PFS, GitLFS, Datapackages, ckan, among others exist. These tools and services allow users to manage large datasets and version-control them. From the point of view of Popper, this is just another tool that will get invoked as part of the execution of a pipeline. As part of our documentation, we have examples on how to use datapackages, and another on how to use data.world.

7.3 How can Popper capture more complex workflows? For example, automatically restarting failed tasks?

A Popper pipeline is a simple sequence of "containerized bash scripts". Popper is not a replacement for scientific workflow engines, instead, its goal is to capture the highest-most workflow: the human interaction with a terminal.

7.4 Can I follow Popper in computational science research, as opposed to computer science?

Yes, the goal for Popper is to make it a domain-agnostic experimentation protocol. See the https://github.com/popperized/popper-examples repository for examples.

7.5 How to apply the Popper protocol for applications that take large quantities of computer time?

The `popper run` takes an optional `STEP` argument that can be used to execute a workflow up to a certain step. Run `popper run --help` for more.

CHAPTER 8

Contributing

Read the CONTRIBUTING.md file contained in the main repository.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search