spline

Release 1.12

May 10, 2021

Contents:

CHAPTER 1

Motivation

Working a longer time with tools like Jenkins and Travis CI you might find out that you loose a lot of time because of try and error. You change the pipeline on a feature branch, push it remote and then run the pipeline analyzing the results. As an example you cannot easily use a Jenkinsfile locally since that Groovy code does use a so called DSL accessing Jenkins and the plugin infrastructure in a running Jenkins instance.

I have been seeking for a better solution where you can do most things already on your own machine. Basically all concepts like matrix, stages and parallelism should be available in a simple terminal (console).

Also it allows using this tool in different existing environments like Jenkins and Travis CI where you can keep the Jenkinsfile (.travis.yml) very simple and short while your pipeline definition yaml contains all.

CHAPTER 2

Quickstart

2.1 Usage

That installs the spline tool including all of its dependencies:

```
pip install spline
```

When you have a pipeline definition (example: pipeline.yaml) then you can run it with:

```
spline --definition=pipeline.yaml
```

Some simple examples you can see in the example folder of the project repository and also spline itself provides that file (exception: Docker tests are skipped).

The minimum structure of a pipeline definition file should look like following:

```
pipeline:
    - stage(Example):
        - tasks(ordered):
            - shell:
                script: echo "hello world!"
```

The output:

```
$ spline --definition=minimum.yaml
2017-11-18 11:24:25,875 - spline.application - Running with Python 2.7.13 (default,␣
↪Jan 19 2017, 14:48:08) [GCC 6.3.0 20170118]
2017-11-18 11:24:25,883 - spline.application - Running on platform Linux-4.9.0-3-
↪amd64-x86_64-with-debian-9.1
2017-11-18 11:24:25,884 - spline.application - Processing pipeline definition
↪'minimum.yaml'
2017-11-18 11:24:25,908 - spline.application - Schema validation for 'minimum.yaml'␣
↪succeeded
2017-11-18 11:24:25,934 - spline.components.stage - Processing pipeline stage 'Example
↪'
```

```
2017-11-18 11:24:25,934 - spline.components.tasks - Processing group of tasks
2017-11-18 11:24:25,934 - spline.components.tasks - Processing Bash code: start
2017-11-18 11:24:25,942 - spline.components.bash - Running script /tmp/pipeline-
↪script-D3N3F9.sh
2017-11-18 11:24:25,948 - spline.components.tasks -  | hello world!
2017-11-18 11:24:25,949 - spline.components.tasks -  |
2017-11-18 11:24:25,950 - spline.components.bash - Exit code has been 0
2017-11-18 11:24:25,950 - spline.components.tasks - Processing Bash code: finished
```

2.2 Development

```
git clone https://github.com/Nachtfeuer/pipeline.git
cd pipeline
./unittests.sh
# OR tox -e py35 OR tox -e py36 (see tox.ini)
tox -e py27
```

For the purpose to test a specific Python version that does not exist on your system you can choose one of following commands:

```
spline --definition=pipeline.yaml --matrix-tags=py27
spline --definition=pipeline.yaml --matrix-tags=py33
spline --definition=pipeline.yaml --matrix-tags=py34
spline --definition=pipeline.yaml --matrix-tags=py35
spline --definition=pipeline.yaml --matrix-tags=py36
spline --definition=pipeline.yaml --matrix-tags=pypy
spline --definition=pipeline.yaml --matrix-tags=pypy3
```

If you leave away those tags it will run for all Python versions. A special note on Python 3.6.x: I have provided two ways to deal with it:

- *init_py36_compile*: A concrete package is downloaded and built. You have to rename it to *init_py36* to use it (rename the other).
- *init_py36*: It currently download ready made CentOS 7 packages; you save time with it.

The *init_py36* is the name that is constructed via the matrix so ensure you have the one you prefer.

CHAPTER 3

Real Example

3.1 Python and tox

Like for Java using Maven or Gradle or using CMake for C++ is tox a tool for Python. It does simplify the support for multiple Python version and the quite comfortable description of the commands and its environments. The spline project has a complete demo project for Python in folder **examples/python/primes**.

3.2 Quickstart

You require spline >= 1.2. It's possible to run tox without parameters but then you need to have all listed Python versions installed. I usually have Python 2.7.x and Pyton 3.5.x on my machine so I could test the project like following: **-e py27 -e py35**.

```
pip install spline tox --upgrade
git clone https://github.com/Nachtfeuer/pipeline.git
cd pipeline/examples/python/primes
tox -e py27
```

The tox.ini covers:

- pep8 (tox -e pep8)
- pep257 (tox -e pep257)
- pylint (tox -e pylint)
- flake8 (tox -e flake8)
- radon (tox -e radon)
- nosetests (tox -e tests, tests with pyhamcrest, 100% coverage as limit)
- packaging (tox -e package wheel file)

Using commands like **tox -e radon** it does use the Python version on your host.

3.3 Spline and matrix build

However the different Python versions will introduce different behavior (often) so you constantly have to verify. The spline tool does help you with this by isolating builds into Docker containers; with this you can test even locally **all** Python version also you have just one Python version on your machine.

So let's start with the matrix definition:

```
matrix:
- name: Python 2.7
    env: {PYTHON_VERSION: py27}
    tags: ['py27']
- name: Python 3.5
    env: {PYTHON_VERSION: py35}
    tags: ['py35']
```

Keeping it simple (demo) I just defined a few Python versions but with given examples it's pretty easy to add more. The given setup will inject the environment variable **PYTHON_VERSION** to be used as filter for the templates in the model. The tags are provided to allow filtering for one concrete Python version only.

3.4 The model

The next step is to define a **model**:

```
model:
    templates:
        init_py27: |
            yum -y install centos-release-scl yum-utils git
            yum-config-manager --enable rhel-server-rhscl-7-rpms
            yum -y install python27
            scl enable python27 "bash -c \"pip install setuptools --upgrade\""
            scl enable python27 "bash -c \"pip install tox\""
            scl enable python27 "bash -c \"{{ env.PIPELINE_BASH_FILE }} RUN\""
```

The Python 3.5 part is also contained (see pipeline.yaml). The main point here to understand is that **scl enable** does use a mechanism where you have to specify a script that is executed **in context** of the specified environent (here: python27). The variable **PIPELINE_BASH_FILE** is generated (injected) by the **spline** tool. You either can refer to by $ syntax (Bash way) or using Jinja2 syntax (as done here).

3.5 The init part of the script

The Bash script that is calling your code running inside a Docker container is called first time with the parameter **INIT**. The Bash case structure handles that rendering the Python template we need for the currently running matrix; so we have to fetch exactly that template from the model which relates to current **PYTHON_VERSION**. Because the template also contains Jinja2 code we have to apply the **render** filter passing the environment variables. The template (last line of it) does call the build script again but now with parameter **RUN** which gives you the possibility to implement your build process inside the Docker container **and** inside the correct Python environment.

```
- docker(container):
    mount: yes
    script: |
      case $1 in
        INIT)
```

```
          {{ model.templates['init_'+env.PYTHON_VERSION]|render(env=env) }}
          ;;
        RUN)
          echo "Running build with $(python -V)"
          ;;
```

3.6 The run part of the script

Of course we don't print just the Python version (as shown before); the final **RUN** case looks like following:

```
RUN)
    echo "Running build with $(python -V)"
    mkdir /build

    # copying all files under version into the container
    pushd /mnt/host/examples/python/primes
    tar cvzf /build/demo.tar.gz $(git ls-files)
    popd

    pushd /build
    # unpacking the copied sources files
    tar xvzf demo.tar.gz
    rm -f demo.tar.gz
    # running the build
    tox -e {{ env.PYTHON_VERSION }}
    popd
    ;;
```

We are inside the Docker container and also running in context of a concrete Python version. Now a build folder will be generated where we place the Python code. It's not optimal to run directly on the shared workspace (repository) because:

- The Docker standard user is root and generated files and folders on the Docker host probably raise permission issues when it comes to cleanup. Yes you can organize to be same user as in the host but with some effort (my personal opinion: avoid it).
- If you run in parallel you share folders even when they are temporary build output (my personal opinion: avoid it).
- On some systems the exchange of files and folders on those Docker mounts is expensive.

That's why I have choosen the variant to use Git since Git exactly knows all files (and folders) under versions copying it into the build folder of the Docker container. After unpacking you simply call **tox -e {{ env.PYTHON_VERSION }}** and your build runs fully isolated inside the Docker container.

The last lines (I don't print all - too many lines) look like following:

```
2017-12-10 11:50:06,230 - spline.components.tasks -  | creating build/bdist.linux-x86_
↪64/wheel/pipeline_demo_python_primes-1.0.dist-info/WHEEL
2017-12-10 11:50:06,230 - spline.components.tasks -  | ___________________________________
↪____ summary ____________________________________
2017-12-10 11:50:06,230 - spline.components.tasks -  |   py27: commands succeeded
2017-12-10 11:50:06,230 - spline.components.tasks -  |   congratulations :)
...
2017-12-10 11:51:24,231 - spline.components.tasks -  | creating build/bdist.linux-x86_
↪64/wheel/pipeline_demo_python_primes-1.0.dist-info/WHEEL
2017-12-10 11:51:24,231 - spline.components.tasks -  | ___________________________________
↪____ summary ____________________________________
```

```
2017-12-10 11:51:24,232 - spline.components.tasks -   |   py35: commands succeeded
2017-12-10 11:51:24,232 - spline.components.tasks -   |   congratulations :)
```

3.7 Run the build (without matrix filtering)

Remains to show how the matrix build is usually executed. For the demo inside the spline repository you have to be in the root of it (because git requires .git from mount):

```
spline --definition=examples/python/primes/pipeline.yaml
```

3.8 Run the build (with a matrix filter)

If you would like to run one Python version only you can use **–matrix-tags**. It accepts a comma separated list of tag names. In given example we run the whole pipeline for Python 3.5.x only.

```
spline --definition=examples/python/primes/pipeline.yaml --matrix-tags=py35
```

Here we run the whole pipeline for Python 2.7.x and Python 3.5.x:

```
spline --definition=examples/python/primes/pipeline.yaml --matrix-tags=py27,py35
```

3.9 Matrix build in Travis CI

The option **–matrix-tags** is also very probably of interest when using it in matrix builds with Travis CI (extract of a .travis.yml file):

```
env:
    matrix:
    - PYTHON_VERSION=py27
    - PYTHON_VERSION=py33
    - PYTHON_VERSION=py34
    - PYTHON_VERSION=py35

script: spline --definition=pipeline.yaml --matrix-tags=${PYTHON_VERSION}
```

3.10 Some final notes

- For the moment it seems that the output of one Bash execution is passed back to the called after finish of it which results in a delay until you see something. I have filed an issue: #28: Asynchronous Bash execution. When I find a solution then I will remove this point.
- If you copy back things into workspace (mount) keep in mind to use **chown -R ${UID}:${GID} <path or file>**.

CHAPTER 4

The pipeline

The pipeline is a list of stages. It also may have environment blocks.

```
pipeline:
- env:
    mode: test

- stage(one):
    - tasks(ordered):
        - shell:
            script: echo "{{ env.mode }}: script one"

- stage(two):
    - tasks(ordered):
        - shell:
            script: echo "{{ env.mode }}: script two"
```

CHAPTER 5

The Pipeline matrix

5.1 Usage

A matrix basically has a name and assigned environment variables. The purpose is to support that same pipeline can run for different parameters. Examples are running with different compilers, interpreters or databases. In addition you can specify tags which allow to filter for certain matrix runs.

```
matrix:
    - name: one
      env:
        mode: one
      tags:
        - first

    - name: two
      env:
        mode: two
      tags:
        - second
```

With this example you can filter for second matrix item like this:

```
pipeline --definition=example.yaml --matrix-tags=second
```

5.2 Parallelization

While **matrix** as well as **matrix(ordered)** are representing ordered pipeline execution you also can specify **matrix(parallel)**. Using parallel all specified matrix items (pipeines) are running in parallel. Parallel matrixs and parallel tasks can be combined.

Be aware that parallelization works just as good as many cpu you have and as less competition.

CHAPTER 6

Pipeline stages

Each stage is a list of tasks blocks. It also may have environment blocks.

```
- stage(one):
    - env:
        mode: test

    - tasks(ordered):
        - shell:
            script: echo "{{ env.mode }}: script one"

    - tasks(ordered):
        - shell:
            script: echo "{{ env.mode }}: script two"
```

The stage name in the round brackets can be any text. It's assumed that a stage should reflect the individual phases of the a CI/CD pipeline including (unordered):

- preparation
- build
- unittests
- static code analysis
- packaging
- integration/regression tests
- image creation (docker, AWS, . . .)
- deployment

CHAPTER 7

The Shell

7.1 One line

The shell is a yaml definition for executing a Bash script.

```
- shell:
    script: echo "hello world!"
```

As an alternative when given script content is a valid path and filename of an existing Bash script then those one will be taken. Please note that the content of each script is copied into a temporary one and executed.

7.2 Multipe lines

You also can have multiple lines:

```
- shell:
    script: |
        echo "hello world 1!"
        echo "hello world 2!"
```

7.3 Jinja templating supported

Jinja templating is supported. Currently two variables are available:

env

Gives you access to the environment variables as defined when the spline tool has been started; in addition you can add environment variables or overwrite existing ones. Please note that the value is always a string.

model

The model is a dictionary (map) with keys and the values can be any valid yaml construct that results in a valid Python data hierarchy.

variables

You can specify a field **variable** on each shell and the output of the Bash will be stored under the defined name. However a special note on this: when you define a task block for parallel tasks then one task cannot access a variable by another parallel task in same execution block; but when such tasks are separated by an **env** entry each task after that entry is able to use it also those run in parallel too. More on this you can read in the chapter about tasks.

Here's a simple example for the access:

```
- tasks:
    - env:
        count: "3"

    - shell:
        script: echo "{{ env.USER }}"
        variable: user

    - shell:
        script: |
            {% for c in range(env.count|int) %}
            echo "{{ c+1 }}:{{ env.message }}"
            {% endfor %}
            echo "USER={{ variables.user }}"
            echo "foo={{ model['foo'] }}"
```

More details on **env** and **model** you can see in a separate chapter.

7.4 Tags

Finally you can specify tags:

```
- shell:
    script: echo "hello world!"
    tags:
        - simple
- shell:
    script: echo "hello world!"
    tags:
        - test
```

Executing the spline tool you can specify **–tags=test** which executes shells only with given tag. You also can specify a comma separted list of tags to allow more shells: **–tags=test,simple**

One usecase might be to isolate a shell for testing purpose.

7.5 "With" attribute

Using the **with** attribute you can run same task as often as many entries you provide. The entries are representing a list but the item can be any valid yaml structure; in the example a dictionary is used:

```
- shell:
    script: |
        echo "{{ item.message }}: start"
        sleep {{ item.time }}
        echo "{{ item.message }}: done"
    with:
        - message: first
          time: 3
        - message: second
          time: 2
        - message: third
          time: 1
```

You also can use a **rendered with** like following when you put the list of items into the model:

```
- shell:
    script: echo "{{ item }}"
    with: "{{ model.data }}"
```

Finally all generated tasks (shell or docker container) are added to the list of tasks to be processed and it depends on the setup of the **tasks** block whether those tasks are executed in **order** or in **parallel**. Please have a look and try the example **with.yaml** in the repository.

7.6 Colors

Colors are working fine!

```
- shell:
    script: |
        echo -e "\e[31mRed World\e[0m"
        echo -e "\e[33mOrange World\e[0m"
        echo -e "\e[34mBlue World\e[0m"
        echo -e "\e[35mMagenta World\e[0m"
```

7.7 Conditional tasks

The field **when** allows you to define a condition; when evaluated as true then the task is executed otherwise not. More details you can read in the separate section *Conditional Tasks*.

CHAPTER 8

The Python task

Python behaves *pretty the same way* as a normal bash script except that the code goes through the Python interpreter found in the search path:

```
model:
    message: 'hello world'

pipeline:
    - stage(Example):
        - tasks(ordered):
            - python:
                script: |
                    import sys
                    print(sys.version.replace("\n", ""))
                    print("{{ model.message }}{{ item }}!")
                with:
                    - 1
                    - 2
                    - 3
```

Of course you can use Jinja2 templating accessing:

- the model
- and the environment variables
- optional the item variable when using the **width** field.
- optional you access a variable when generated by a previous task. (already demonstrated when explaining the shell)

Also tags are allowed and you can specify a title for logging.

CHAPTER 9

The Model

9.1 Introduction

The model is a flexible way to define data. For the moment you can define it only once at global level:

```
model:
    max-number: 100

pipeline:
    - stage(Calculate Primes):
        - tasks(ordered):
            - shell:
                script: |
                    function is_prime() {
                        n=$1
                        if [ "${n}" -lt 2 ]; then return; fi
                        if [ "$(($n % 2))" -eq 0 ]; then
                            if [ "${n}" == "2" ]; then echo "yes"; fi
                            return;
                        fi
                        d=$(echo "sqrt(${n})"|bc)
                        for k in $(seq 3 2 ${d}); do
                            if [ "$(($n % $k))" -eq 0 ]; then return; fi
                        done
                        echo "yes"
                    }

                    for n in $(seq 0 {{ model['max-number'] }}); do
                        if [ "$(is_prime ${n})" == "yes" ]; then
                            echo -n "${n} ";
                        fi
                    done
                tags:
                    - embedded
```

The output looks like following:

```
$ spline --definition=examples/primes.yaml --tags=embedded
2017-11-20 05:53:45,150 - spline.application - Running with Python 2.7.13 (default,␣
↪Jan 19 2017, 14:48:08) [GCC 6.3.0 20170118]
2017-11-20 05:53:45,177 - spline.application - Running on platform Linux-4.9.0-3-
↪amd64-x86_64-with-debian-9.1
2017-11-20 05:53:45,177 - spline.application - Processing pipeline definition
↪'examples/primes.yaml'
2017-11-20 05:53:45,210 - spline.application - Schema validation for 'examples/primes.
↪yaml' succeeded
2017-11-20 05:53:45,214 - spline.components.stage - Processing pipeline stage
↪'Calculate Primes'
2017-11-20 05:53:45,214 - spline.components.tasks - Processing group of tasks
2017-11-20 05:53:45,215 - spline.components.tasks - Processing Bash code: start
2017-11-20 05:53:45,220 - spline.components.bash - Running script /tmp/pipeline-
↪script-i6l5rx.sh
2017-11-20 05:53:46,261 - spline.components.tasks -  | 2 3 5 7 11 13 17 19 23 29 31␣
↪37 41 43 47 53 59 61 67 71 73 79 83 89 97
2017-11-20 05:53:46,261 - spline.components.bash - Exit code has been 0
2017-11-20 05:53:46,262 - spline.components.tasks - Processing Bash code: finished
```

As an alternative you also can do it like following:

```
model:
    max-number: 100

pipeline:
    - stage(Calculate Primes):
        - tasks(ordered):
            - shell:
                script: examples/primes.sh {{ model['max-number'] }}
                tags:
                    - file
```

For completeness:

```
$ spline --definition=examples/primes.yaml --tags=file
```

Lists in yaml will be converted into Python lists and yaml dictionaries will be converted into Python dictionaries. All basically as you would expect.

9.2 Nested templates

The model also can be used for storing templates that can be injected into scripts. You probably also would like to pass then the model and environment variables to it:

```
model:
    templates:
        script: |
            echo "{{ model.message }} {{ env.who }}!"
    message: "hello"

pipeline:
    - env:
        who: world
```

```
- stage(Test):
    - tasks(ordered):
        - shell:
            script: "{{ model.templates.script|render(model=model, env=env) }}"
```

That's just a very simple example.

CHAPTER 10

The Environment Variables

```
pipeline:
- env:
    a: "hello"

- stage(Environment Variables):
    - env:
        b: "world"

    - tasks(ordered):
        - env:
            c: "for all"

        - shell:
            script: |
                echo "a=$a"
                echo "b=$b"
                echo "c=$c"
```

An extract from the output when running the pipeline:

```
2017-11-20 18:47:08,209 - spline.components.bash - Running script /tmp/pipeline-
↪script-w2MUih.sh
2017-11-20 18:47:08,215 - spline.components.tasks -   | a=hello
2017-11-20 18:47:08,215 - spline.components.tasks -   | b=world
2017-11-20 18:47:08,216 - spline.components.tasks -   | c=for all
```

All defined variables are merged together:

- first all environment variables on pipeline level are taken
- in the resulting dictionary all environment variables from stage level are used for updating. New variables are added and existing variables are overwritten.
- in the resulting dictionary all environment variables from tasks level are used for updating. New variables are added and existing variables are overwritten.

Please note: All values have to be strings.

In a Bash script you also can refer to the variables using Jinja templating like `{{ env.a }}`.

CHAPTER 11

The Tasks

It's a list of tasks basically meaning a shell as Bash script or runnning inside a Docker container. Tasks can be **ordered** or **parallel**.

11.1 Ordered tasks

Ordered tasks can written as `- tasks:` or as `- tasks(ordered):` (the way you prefer). It means the same: one shell script is executed after the other:

```
- tasks(ordered):
    - shell:
        script: echo "hello world one!"
    - shell:
        script: echo "hello world two!"
```

11.2 Parallel tasks

All tasks are running in parallel as much as possible. The Python module **multiprocessing** is used.

```
- tasks(parallel):
    - shell:
        script: echo "hello world one!"
    - shell:
        script: echo "hello world two!"
```

Please note:

- It's not a good idea to interrupt the pipeline with Ctrl-C because multiprocessing is used.
- Example: When you have 4 cpus but more than 4 tasks it might happen that the tasks do not finish in time constraints as you expect. It seems that one task is assigned to one cpu only at a time.

- When one task fails the pipeline stops after all tasks has been finished.
- When using multiple enviroment blocks tasks run in parallel only between two of those "env" blocks.

11.3 Environment variables

Besides a tasks the list also may contain one or more blocks for environment variables.

```
- env:
    a: "hello"
    b: "world"
```

The last block overwrites the previous one; existing variables are overwriten, new ones are added.

11.4 Variables on tasks

On shells, python scripts and docker container tasks you can specify a variable and variables are stored at pipeline level. When a block of parallel tasks start all variables before this time are passed to the tasks and while those are running new variables cannot be evaluated. But a tasks block also may contain **env** entries so you can split parallel tasks in two (or more) blocks. Each new block is able to access last stored variables; here a rough example:

```
- tasks(parallel):
    # first block
    - shell:
        script: echo "hello"
        variable: one
    - shell:
        script: echo "world"
        variable: two

    - env:
        message: "a great"

    # second block
    - shell:
        script: echo "{{ env.message }} {{ variables.one }} {{ variables.two }}"
    - shell:
        script: echo "{{ env.message }} {{ variables.one }}"
    - shell:
        script: echo "{{ env.message }} {{ variables.two }}"
```

The two commented blocks are executed in order because of an **env** entry inbetween but all tasks of one block are executed in parallel. When executing it looks like following:

```
$ spline --definition=examples/variables.yaml 2>&1 | grep "great"
2018-01-22 19:49:44,576 - spline.components.tasks.worker -  | a great world
2018-01-22 19:49:44,577 - spline.components.tasks.worker -  | a great hello
2018-01-22 19:49:44,581 - spline.components.tasks.worker -  | a great hello world
```

When the tasks are ordered a previous variable by a previous task can be evaluated immediately.

CHAPTER 12

The Docker Container Script

12.1 Simple Example

The Docker container block **is basically the same** as the shell block with the exception that a simple wrapper code is injected for Running the Docker container. Assume following block as an example:

- it runs a Docker container.
- since no image is specified *centos:7* is used (as default)
- after the injected Bash code has finished the Docker container will be automatically removed.

```
- docker(container):
    script: |
        yum -y install epel-release > /dev/null 2>&1
        yum -y install figlet  > /dev/null 2>&1
        figlet -f standard "docker" | sed -e 's: :.:g'
    tags:
        - no-image
```

The code snippet you can find in the tests:

```
$ PYTHONPATH=$PWD python scripts/pipeline --definition=tests/pipeline-015.yaml --
↪tags=no-image
2017-10-29 12:33:59,091 - pipeline.application - Running with Python 2.7.13 (default,␣
↪Jan 19 2017, 14:48:08) [GCC 6.3.0 20170118]
2017-10-29 12:33:59,104 - pipeline.application - Running on platform Linux-4.9.0-3-
↪amd64-x86_64-with-debian-9.1
2017-10-29 12:33:59,104 - pipeline.application - Processing pipeline definition
↪'tests/pipeline-015.yaml'
2017-10-29 12:33:59,135 - pipeline.application - Schema validation for 'tests/
↪pipeline-015.yaml' succeeded
2017-10-29 12:33:59,137 - pipeline.components.stage - Processing pipeline stage 'test'
2017-10-29 12:33:59,137 - pipeline.components.tasks - Processing group of tasks
2017-10-29 12:33:59,138 - pipeline.components.tasks - Processing Bash code: start
2017-10-29 12:33:59,146 - pipeline.components.bash - Running script /tmp/pipeline-
↪script-z3eXdd.sh
```

```
2017-10-29 12:34:16,404 - pipeline.components.tasks -  | ....._............_.........
↪...
2017-10-29 12:34:16,404 - pipeline.components.tasks -  | ..__|.|.___...___|.|._____._.
↪__.
2017-10-29 12:34:16,405 - pipeline.components.tasks -  | ./._`.|/._.\./.__|.|/./._.\.
↪'__|
2017-10-29 12:34:16,405 - pipeline.components.tasks -  | |.(_|.|.(_).|.(__|...<..__/.
↪|...
2017-10-29 12:34:16,405 - pipeline.components.tasks -  | .\__,_|\___/.\___|_|\_\___|_
↪|...
2017-10-29 12:34:16,405 - pipeline.components.tasks -  | ..........................
↪...
2017-10-29 12:34:16,405 - pipeline.components.tasks -  |
2017-10-29 12:34:16,405 - pipeline.components.bash - Exit code has been 0
```

12.2 Specifying an image

You also can specify an image:

```
- docker(container):
    image: centos:7.3.1611
    script: cat /etc/redhat-release
    tags:
        - with-image
```

Here's an extract of the output:

```
2017-10-29 12:46:06,080 - pipeline.components.bash - Running script /tmp/pipeline-
↪script-36Ga0I.sh
2017-10-29 12:46:07,583 - pipeline.components.tasks -  | CentOS Linux release 7.3.
↪1611 (Core)
2017-10-29 12:46:07,583 - pipeline.components.tasks -  |
```

12.3 Using user labels

The **docker(container)** task also have an optional dictionary attribute **lables**. The key has to be upper case and to start with **UL** (user label). All label allow Jinja2 rendering.

The idea is to be able to adjust custom values from outside spline (environment variables) that can be used to query containers more easily.

12.4 How to find a Docker container

- Each Docker container gets additional labels: - **pipeline** - which contains the PID of the pipeline. - **pipeline-stage** - pipeline stage in which the Docker container has been created. - **context** - always "pipeline" - **creator** - the PID of the shell which created the Docker container.
- with those information you have some control for being able to query a concrete container without knowing the Docker container name (you need not worry about container names since Docker does it for you).

- If you create multiple Docker container per stage then (TODO) there will be a label that can be adjusted via the yaml to reduce the query to the right container.
- Have a look at the examples [docker.yaml](examples/docker.yaml).

12.5 Mounts

For good reasons various number of mounts have been minimized to the most essential ones:

- one mount (always) for the script mechanism (you shouldn't care)
- one mount (on demand) if you need to exchange things with the host

The next example does activate the second mount which maps $PWD as */mnt/host* inside the Docker container. Here I write a file to the host and another script dumps it and removes the file afterwards.

```
- docker(container):
    script: |
        echo "hello world" > /mnt/host/hello.txt
        chown ${UID}:$(GID} /mnt/host/hello.txt
    mount: true

- shell:
    script: |
        cat hello.txt
        rm -f hello.txt
```

Please note: Usually the Docker user is root (by default) and when you copy content to the host the caller might fail on removing that files and folders because of missing permissions. That's why the user id and group id is always passed to the container allowing you to adjust the permissions correctly.

12.6 Network

The optional field **network** allows you to specify a network name. You can create a network with *docker network create <name>* (see docker.yaml in examples folder). Another usecase is docker-compose which usually does create a network for all containers it does create; if you now create an additional container that should be able to work with the existing ones you have to "join" the network by specifying the name of that network.

```
- docker(container):
    network: demo
    script: echo "hello world"
```

12.7 "With" attribute

It's exactly the same as for shell - please read the details there.

12.8 Conditional tasks

The field **when** allows you to define a condition; when evaluated as true then the task is executed otherwise not. More details you can read in the separate section *Conditional Tasks*.

CHAPTER 13

The Docker Image Script

13.1 Simple example

The next example demonstrates one way on how to create a docker image for Python 3.6.

```
- docker(image):
    name: python
    tag: "3.6"
    unique: no
    script: |
      FROM centos:7
      RUN yum -y install yum-utils git
      RUN yum -y install https://centos7.iuscommunity.org/ius-release.rpm
      RUN yum -y install python36u python36u-pip
      RUN pip3.6 install tox
```

When you run this (see examples folder for docker-image.yaml) then last lines should look like following:

```
2017-12-22 09:20:15,179 - spline.components.tasks -   | Installing collected packages:␣
↪py, six, pluggy, virtualenv, tox
2017-12-22 09:20:15,339 - spline.components.tasks -   |   Running setup.py install for␣
↪pluggy: started
2017-12-22 09:20:15,665 - spline.components.tasks -   |     Running setup.py install␣
↪for pluggy: finished with status 'done'
2017-12-22 09:20:15,800 - spline.components.tasks -   | Successfully installed pluggy-
↪0.6.0 py-1.5.2 six-1.11.0 tox-2.9.1 virtualenv-15.1.0
2017-12-22 09:20:16,588 - spline.components.tasks -   |  ---> 29abbe7ec073
2017-12-22 09:20:16,601 - spline.components.tasks -   | Removing intermediate␣
↪container 5ceeb0cf5b89
2017-12-22 09:20:16,601 - spline.components.tasks -   | Successfully built 29abbe7ec073
2017-12-22 09:20:16,608 - spline.components.tasks -   | Successfully tagged python:3.6
2017-12-22 09:20:16,610 - spline.components.bash - Exit code has been 0
2017-12-22 09:20:16,611 - spline.components.tasks - Processing Bash code: finished
```

You can verify afterwards:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
python              3.6                 29abbe7ec073        9 minutes ago       605MB
$ docker run --rm -i python:3.6 bash -c "python3.6 -V"
Python 3.6.3
```

13.2 The option “unique”

In the example above **unique** has been set to **no**. The default is **yes** which injects the pid of the spline tool into the name. The idea is to allow multiple images generated in parallel without conflicts.

13.3 Dockerfile

The **script** field represents the Dockerfile and please refer to the official Documentation if you need to know more. However Jinja2 templating is supported:

```
- docker(image):
    name: python
    tag: "3.6"
    unique: no
    script: "{{ model.templates.dockerfiles.py36 }}"
```

Please note: you also can use jinja templaring in the tag.

The compete example - as already mentioned - in the examples folder.

13.4 Conditional tasks

The field **when** allows you to define a condition; when evaluated as true then the task is executed otherwise not. More details you can read in the separate section *Conditional Tasks*.

CHAPTER 14

The Packer Task

14.1 Setup

You have to ensure that the packer tool is installed and in the search path. The installation is simple (mainly download, unpack and copy of one binary).

14.2 Simpe Example

The next example demonstrates how you can use the packer task to generate a Docker image:

```
- packer:
    script: |
        {"builders": [{
            "type": "docker",
            "image": "{{ model.image }}",
            "commit": true,
            "changes": [
                "LABEL pipeline={{ env.PIPELINE_PID }}",
                "LABEL pipeline-stage={{ env.PIPELINE_STAGE }}"
            ]
        }],

        "provisioners": [{
            "type": "shell",
            "inline": [
                "yum -y install python-setuptools",
                "easy_install pip",
                "pip install tox"
            ]
        }],

        "post-processors": [{
```

```
            "type": "docker-tag",
            "repository": "spline/packer/demo",
            "tag": "0.1"
        }]}
```

The output looks like following starting with:

```
2018-03-16 05:20:50,948 - spline.components.bash - Running script /tmp/pipeline-
↪script-5N9ZeH.sh
2018-03-16 05:20:50,955 - spline.components.tasks -  | packer build  /tmp/packer-
↪e7je86.json
2018-03-16 05:20:51,125 - spline.components.tasks -  | docker output will be in this␣
↪color.
2018-03-16 05:20:51,128 - spline.components.tasks -  |
2018-03-16 05:20:51,163 - spline.components.tasks -  | ==> docker: Creating a␣
↪temporary directory for sharing data...
2018-03-16 05:20:51,164 - spline.components.tasks -  | ==> docker: Pulling Docker␣
↪image: centos:7
2018-03-16 05:21:03,703 - spline.components.tasks -  |     docker: 7: Pulling from␣
↪library/centos
2018-03-16 05:21:14,557 - spline.components.tasks -  |     docker: Digest:␣
↪sha256:dcbc4e5e7052ea2306eed59563da1fec09196f2ecacbe042acbdcd2b44b05270
2018-03-16 05:21:14,559 - spline.components.tasks -  |     docker: Status: Image is␣
↪up to date for centos:7
2018-03-16 05:21:14,561 - spline.components.tasks -  | ==> docker: Starting docker␣
↪container...
2018-03-16 05:21:14,564 - spline.components.tasks -  |     docker: Run command:␣
↪docker run -v /home/thomas/.packer.d/tmp/packer-docker809184673:/packer-files -d -i␣
↪-t centos:7 /bin/bash
2018-03-16 05:21:15,115 - spline.components.tasks -  |     docker: Container ID:␣
↪2543f16b4acc3e107ef7ce5b1e8164d66bfbc0a0a34ad682c3b75db390677e80
2018-03-16 05:21:15,198 - spline.components.tasks -  | ==> docker: Provisioning with␣
↪shell script: /tmp/packer-shell164186494
2018-03-16 05:21:16,627 - spline.components.tasks -  |     docker: Loaded plugins:␣
↪fastestmirror, ovl
2018-03-16 05:21:25,764 - spline.components.tasks -  |     docker: Determining␣
↪fastest mirrors
2018-03-16 05:21:27,312 - spline.components.tasks -  |     docker:  * base: centos.
↪intergenia.de
2018-03-16 05:21:27,316 - spline.components.tasks -  |     docker:  * extras: centos.
↪intergenia.de
2018-03-16 05:21:27,319 - spline.components.tasks -  |     docker:  * updates: mirror.
↪fra10.de.leaseweb.net
2018-03-16 05:21:30,280 - spline.components.tasks -  |     docker: Resolving␣
↪Dependencies
2018-03-16 05:21:30,281 - spline.components.tasks -  |     docker: --> Running␣
↪transaction check
2018-03-16 05:21:30,282 - spline.components.tasks -  |     docker: ---> Package␣
↪python-setuptools.noarch 0:0.9.8-7.el7 will be installed
```

and finishing with:

```
2018-03-16 05:21:39,991 - spline.components.tasks -  | ==> docker: Committing the␣
↪container
2018-03-16 05:21:41,831 - spline.components.tasks -  |     docker: Image ID:␣
↪sha256:270dbc58a828269a069142c8cef9c7d93c735b9217d617ee123cd5c4e2d552a2
2018-03-16 05:21:41,832 - spline.components.tasks -  | ==> docker: Killing the␣
↪container: 2543f16b4acc3e107ef7ce5b1e8164d66bfbc0a0a34ad682c3b75db390677e80
```

```
2018-03-16 05:21:42,380 - spline.components.tasks -  | ==> docker: Running post-
↪processor: docker-tag
2018-03-16 05:21:42,385 - spline.components.tasks -  |     docker (docker-tag):␣
↪Tagging image:␣
↪sha256:270dbc58a828269a069142c8cef9c7d93c735b9217d617ee123cd5c4e2d552a2
2018-03-16 05:21:42,385 - spline.components.tasks -  |     docker (docker-tag):␣
↪Repository: spline/packer/demo:0.1
2018-03-16 05:21:42,451 - spline.components.tasks -  | Build 'docker' finished.
2018-03-16 05:21:42,452 - spline.components.tasks -  |
2018-03-16 05:21:42,452 - spline.components.tasks -  | ==> Builds finished. The␣
↪artifacts of successful builds are:
2018-03-16 05:21:42,455 - spline.components.tasks -  | --> docker: Imported Docker␣
↪image: sha256:270dbc58a828269a069142c8cef9c7d93c735b9217d617ee123cd5c4e2d552a2
2018-03-16 05:21:42,457 - spline.components.tasks -  | --> docker: Imported Docker␣
↪image: spline/packer/demo:0.1
2018-03-16 05:21:43,344 - spline.components.bash - Exit code has been 0
2018-03-16 05:21:43,345 - spline.components.tasks - Processing Bash code: finished
```

14.3 Important notes

- **you don't require packer variables**: Because you directly can use Jinja2 templating you don't require variables in the Packer script.
- **You are responsible**: It depends on you what you are generating and how you do it. The example is just for Docker but Packer does support more image types. Spline does not know how to cleanup things here. Also you have to ensure unique names (if wanted) considering builds running in parallel avoiding any conflicts. The *docker(image)* task (as comparison) injects the spline pid into the image name; you can do it easily using Jinja2 templating but you have to do it yourself.
- **No filter**: You can have multiple builders in packer and when use them the packer task generates all images (*-only* and *-except* options are not used).
- Packer is enabled for parallelization
- When the build does fail Packer does the cleanup.
- The spline option *–debug* will be passed as *-debug* to the **packer build** command. **Please pay attention here**: you have to press enter for individual steps.

CHAPTER 15

The Ansible(simple) Task

The Ansible task provides you a subset of Ansible; mainly the focus is to have an inventory file (hosts) and one playbook maintainable by one spline document.

15.1 Example

The next example you can see in the folder with same name in file *ansible-docker.yaml*. One Docker container is organized to have **sshd** installed and the Ansible connects to that container installing a few packages.

```
- ansible(simple):
    inventory: |
        [all]
        localhost ansible_host={{ variables.container_host }} ansible_port=22 ansible_
↪connection=ssh ansible_ssh_user=root ansible_ssh_pass={{ env.PASS }}
    limit: all
    script: |
    - hosts: all
        tasks:
        - name: Install packages
            yum:
            name: "{%raw%}{{ item }}{%endraw%}"
            state: present
            with_items:
            {% for package in model.packages %}
              - {{ package }}
            {% endfor %}
```

15.2 Notes on Jinja Templating

Spline does use Jinja2 for templating and Ansible does it as well. You have to control when the templating applies and when not. In given example we would like to have a playbook that finally looks like following:

```
- hosts: all
    tasks:
    - name: Install packages
        yum:
        name: "{{ item }}"
        state: present
        with_items:
          - curl
          - git
          - cmake
```

That's why the evaluation of **item** has been suppressed. Also you cannot insert the packages just by writing **{{ model.packages }}** but the result is a Python list with 3 strings. Three lines with items are wanted (as you can see above); of course a filter could apply rendering as yaml syntax but then you also have to manage indentation which turned out to be difficult (have not found a way to pass current indentation).

15.3 Hosts, ports, user and password

For the Docker example it was sufficient to define user and password in the inventory file. Of course credentials in code are not fine but you can inject the credential from outside (environment variable) or you use the mechanism only for setting up a regression environment with no access to any production environment.

Reading the documentation about Ansible inventory you also shoud be able to use **ansible_ssh_private_key_file**.

Please read: http://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html

CHAPTER 16

Conditional Tasks

16.1 Introduction

Conditional tasks allow you to run certain tasks when the defined condition evaluates to true only. At the moment you can use such conditions on each task: shell, docker(container), docker(image) and python.

16.2 Data sources

There are three sources of information that can be used the moment (via Jinja templating):

- model variables - constant definition in the yaml file.
- task variables - see different type of tasks using the field **variable**
- environment variables - see the **env** entry usable on **matrix**, **pipeline**, **stage** and **tasks**

16.3 Rules

You have to comply some rules when using conditions. Following variants of conditions are intended:

- *{{ variables.cpu_count }} == 1* - comparison of two integers to be equal
- *not {{ variables.cpu_count }} == 1* - comparison of two integers to be not equal
- *{{ variables.cpu_count }} > 1* - comparison of one integer to be greater than another
- *{{ variables.cpu_count }} >= 1* - comparison of one integer to be greater or equal than another
- *{{ variables.cpu_count }} < 2* - comparison of one integer to be less than another
- *{{ variables.cpu_count }} <= 2* - comparison of one integer to be less or equal than another
- *“{{ env.BRANCH_NAME }}” == “master”* - comparison of two strings to be equal

- *not "{{ env.BRANCH_NAME }}" == "master"* - comparison of two strings to be not equal
- *{{ variables.cpu_count }} in [1, 2]* - integer contained in a list of integers
- *{{ variables.cpu_count }} not in [1, 2]* - integer not ontained in a list of integers
- *"{{ env.BRANCH_NAME }}" in ["master", "release"]* - comparison contained in a list of strings
- *"{{ env.BRANCH_NAME }}" not in ["master", "release"]* - comparison not contained in a list of strings

Please note: all other combination that might work should **not** be considered. Future versions of the spline tool will improve the condition checks to be more strict.

Please note: When the jinja templating finally produces a condition with wrong syntax each thrown exception will evaluate the related condition to false. Please check the logs for details then.

16.4 Examples

You can see examples in the file *conditions.yaml* of the tool repository; here is an extract of it:

```
- shell:
    script: echo "integer in integer list comparison"
    when: "{{ model.intval }} in [1234, 4321]"
- shell:
    # task output should not be shown
    script: echo "integer not in integer list comparison"
    when: "{{ model.intval }} not in [1234, 4321]"
```

CHAPTER 17

Hooks

Hooks are defined at same root level as the pipeline or the matrix.

17.1 The cleanup hook

It's basically same as for a shell script with a few differences only:

- When the pipeline succeeds all variables from pipeline level are available.
- When a shell script fails all variables on that level are available
- Additionally the variable **PIPELINE_RESULT** can have the value **SUCCESS** or **FAILURE**.
- Additionally the variable **PIPELINE_SHELL_EXIT_CODE** has the shell exit code of the failed shell or 0 (default)

```
hooks:
    cleanup:
        script: |
            echo "cleanup has been called!"
            echo "${message}"
            echo "PIPELINE_RESULT=${PIPELINE_RESULT}"
            echo "PIPELINE_SHELL_EXIT_CODE=${PIPELINE_SHELL_EXIT_CODE}"
```

CHAPTER 18

The include statement

18.1 Basic Usage

You can use the **!include** statement on maps and lists. You have to ensure that the final document structure is still a valid spline document. Spline will run the validation after the include has been done.

‘‘‘yaml model: !include library/model.yaml pipeline:

- stage(Setup): !include library/setup.yaml
- stage(Build): !include library/build.yaml
- **stage(Test):**
  - !include library/setup-test.yaml
  - !include library/run-test.yaml
  - !include library/teardown-test.yaml
- stage(Deploy): !include library/deploy.yaml

‘‘‘

18.2 Notes

- The loader is evaluating the **!include** statement for the main document

only (by intention).

- the specified file has to exist!

CHAPTER 19

The Even logging

With the command line option *–event-logging* you enable additional logging that measures execution time on the whole application, each pipeline, stage, tasks and docker/shell level.

```
$ PYTHONPATH=$PWD python scripts/pipeline --definition=examples/docker.yaml  --
↪tags=using-mount --event-logging
2017-11-03 05:10:52,742 - pipeline.application - Running with Python 2.7.13 (default,␣
↪Jan 19 2017, 14:48:08) [GCC 6.3.0 20170118]
2017-11-03 05:10:52,757 - pipeline.application - Running on platform Linux-4.9.0-3-
↪amd64-x86_64-with-debian-9.1
2017-11-03 05:10:52,757 - pipeline.application - Processing pipeline definition
↪'examples/docker.yaml'
2017-11-03 05:10:52,824 - pipeline.application - Schema validation for 'examples/
↪docker.yaml' succeeded
2017-11-03 05:10:52,841 - pipeline.components.stage - Processing pipeline stage
↪'example'
2017-11-03 05:10:52,842 - pipeline.components.tasks - Processing group of tasks
2017-11-03 05:10:52,842 - pipeline.components.tasks - Processing Bash code: start
2017-11-03 05:10:52,876 - pipeline.components.bash - Running script /tmp/pipeline-
↪script-Ws20v5.sh
2017-11-03 05:10:54,173 - pipeline.components.tasks -  |
2017-11-03 05:10:54,173 - pipeline.components.bash - Exit code has been 0
2017-11-03 05:10:54,174 - pipeline.components.bash.event - Succeeded - took 1.331764␣
↪seconds.
2017-11-03 05:10:54,174 - pipeline.components.tasks - Processing Bash code: finished
2017-11-03 05:10:54,174 - pipeline.components.tasks - Processing Bash code: start
2017-11-03 05:10:54,176 - pipeline.components.bash - Running script /tmp/pipeline-
↪script-sydNg5.sh
2017-11-03 05:10:54,191 - pipeline.components.tasks -  | hello world
2017-11-03 05:10:54,191 - pipeline.components.tasks -  |
2017-11-03 05:10:54,191 - pipeline.components.bash - Exit code has been 0
2017-11-03 05:10:54,191 - pipeline.components.bash.event - Succeeded - took 0.017044␣
↪seconds.
2017-11-03 05:10:54,192 - pipeline.components.tasks - Processing Bash code: finished
2017-11-03 05:10:54,192 - pipeline.components.tasks.event - Succeeded - took 1.349966␣
↪seconds.
```

```
2017-11-03 05:10:54,192 - pipeline.components.stage.event - Succeeded - took 1.350690␣
↪seconds.
2017-11-03 05:10:54,192 - pipeline.pipeline.event - Succeeded - took 1.351264 seconds.
2017-11-03 05:10:54,192 - pipeline.application.event - Succeeded - took 1.450534␣
↪seconds.
```

CHAPTER 20

Command Line Options

20.1 Dry run mode

Using the option **–dry-run** you get a tool that help you to analyse your pipeline with following rules when the option is set:

- parallelism (matrix and tasks) is disabled
- custom logging is disabled
- the default logging is adjusted to have no timestamps
- using **docker(image)** tasks the Dockerfile is printed as Bash comment; the Dockerfile is not written as a file.
- The cleanup hooks are also not executed but logged.

As an example a **docker(image)** task might look similar to following output:

```
$ spline --definition=examples/docker-image.yaml --dry-run
spline.application - Running with Python 2.7.13 (default, Jan 19 2017, 14:48:08) [GCC␣
↪6.3.0 20170118]
spline.application - Running on platform Linux-4.9.0-3-amd64-x86_64-with-debian-9.1
spline.application - Processing pipeline definition 'examples/docker-image.yaml'
spline.application - Schema validation for 'examples/docker-image.yaml' succeeded
spline.components.stage - Processing pipeline stage 'Example'
spline.components.tasks - Processing group of tasks (parallel=disabled)
spline.components.tasks - Processing Bash code: start
spline.components.bash - Dry run mode for script /tmp/pipeline-script-TRd8fF.sh
spline.components.tasks -  | #!/bin/bash
spline.components.tasks -  | # Dockerfile:
spline.components.tasks -  | # >>
spline.components.tasks -  | # FROM centos:7
spline.components.tasks -  | # RUN yum -y install yum-utils git
spline.components.tasks -  | # RUN yum -y install https://centos7.iuscommunity.org/
↪ius-release.rpm
spline.components.tasks -  | # RUN yum -y install python36u python36u-pip
spline.components.tasks -  | # RUN pip3.6 install tox
```

```
spline.components.tasks -   | #
spline.components.tasks -   | # <<
spline.components.tasks -   | # for visibility in logging
spline.components.tasks -   | echo "docker build -t python:3.6 < dockerfile.dry.run.
↪see.comment"
spline.components.tasks -   | # trying to build docker image
spline.components.tasks -   | docker build -t python:3.6 - < dockerfile.dry.run.see.
↪comment
spline.components.tasks -   | docker_error=$?
spline.components.tasks -   | # cleanup
spline.components.tasks -   | rm -f
spline.components.tasks -   | # give back result
spline.components.tasks -   | exit ${docker_error}
spline.components.tasks - Processing Bash code: finished
```

20.2 Debug

The option *–debug* adjust the Bash option *set -x* which activates the debug mode in Bash. The primes example in the spline repository gives you a good example. I'm just printing the first 20 lines:

```
$ spline --definition=examples/primes.yaml --debug 2>&1 | head -20
2018-01-05 19:31:12,023 - spline.application - Running with Python 2.7.13 (default,␣
↪Jan 19 2017, 14:48:08) [GCC 6.3.0 20170118]
2018-01-05 19:31:12,028 - spline.application - Running on platform Linux-4.9.0-3-
↪amd64-x86_64-with-debian-9.1
2018-01-05 19:31:12,028 - spline.application - Current cpu count is 4
2018-01-05 19:31:12,029 - spline.application - Processing pipeline definition
↪'examples/primes.yaml'
2018-01-05 19:31:12,032 - spline.application - Schema validation for 'examples/primes.
↪yaml' succeeded
2018-01-05 19:31:12,032 - spline.components.stage - Processing pipeline stage
↪'Calculate Primes'
2018-01-05 19:31:12,033 - spline.components.tasks - Processing group of tasks␣
↪(parallel=no)
2018-01-05 19:31:12,033 - spline.components.tasks - Processing Bash code: start
2018-01-05 19:31:12,043 - spline.components.bash - Running script /tmp/pipeline-
↪script-5iRCsz.sh
2018-01-05 19:31:12,050 - spline.components.tasks -   | ++ seq 0 100
2018-01-05 19:31:12,051 - spline.components.tasks -   | + for n in $(seq 0 100)
2018-01-05 19:31:12,052 - spline.components.tasks -   | ++ is_prime 0
2018-01-05 19:31:12,052 - spline.components.tasks -   | ++ n=0
2018-01-05 19:31:12,052 - spline.components.tasks -   | ++ '[' 0 -lt 2 ']'
2018-01-05 19:31:12,052 - spline.components.tasks -   | ++ return
2018-01-05 19:31:12,053 - spline.components.tasks -   | + '[' '' == yes ']'
2018-01-05 19:31:12,053 - spline.components.tasks -   | + for n in $(seq 0 100)
2018-01-05 19:31:12,053 - spline.components.tasks -   | ++ is_prime 1
2018-01-05 19:31:12,053 - spline.components.tasks -   | ++ n=1
2018-01-05 19:31:12,053 - spline.components.tasks -   | ++ '[' 1 -lt 2 ']'
```

20.3 Temporary Scripts Path

The Python library functionality related to temporary folders is explained here: https://docs.python.org/2/library/tempfile.html#tempfile.mkstemp

However you can specify another path that is used to store splines temporary scripts by specifying the path with *–temporary-scripts-path*. When the path doesn't exist the tool tries to create it for you. Here's an example:

```
$ spline --definition=examples/colors.yaml --temporary-scripts-path=$PWD/temp
2018-02-06 05:52:57,547 - spline.application - Running with Python 2.7.13 (default,␣
↪Nov 24 2017, 17:33:09) [GCC 6.3.0 20170516]
2018-02-06 05:52:57,553 - spline.application - Running on platform Linux-4.9.0-5-
↪amd64-x86_64-with-debian-9.3
2018-02-06 05:52:57,553 - spline.application - Current cpu count is 4
2018-02-06 05:52:57,553 - spline.application - Processing pipeline definition
↪'examples/colors.yaml'
2018-02-06 05:52:57,557 - spline.application - Schema validation for 'examples/colors.
↪yaml' succeeded
2018-02-06 05:52:57,558 - spline.components.stage - Processing pipeline stage 'Example
↪'
2018-02-06 05:52:57,559 - spline.components.tasks - Processing group of tasks␣
↪(parallel=no)
2018-02-06 05:52:57,566 - spline.components.tasks - Processing Bash code: start
2018-02-06 05:52:57,568 - spline.components.bash - Running script /work/pipeline/temp/
↪pipeline-script-FxvNFG.sh
```

CHAPTER 21

Unicode

Unicode is not a trivial topic at all.

Even more it can be pretty tricky writing code that does work under multiple Python versions especially when focusing on Python 2.x and Python 3.x.

The reason why I have been investigating into that topic is that I found a build process where a tool has generated console output with unicode characters.

In the example you will find a special-characters.yaml. I tested it for Python 2.7.x and Python 3.5.x on my machine.

Let's see ...

CHAPTER 22

The one file report

22.1 Introduction

At the moment a one file HTML is supported only. On updates (stages) the tool overwrites same file each time which current report data displaying a table showing each matrix and each stage.

- a green cell indicates a successful completed stage
- a red cell indicates a failed stage
- a yellow stage indicates a stage that has not been processed

Information as currently the state (started, succeeded and failed) and the duration.

You enable it by using the command line option *–report* (default: off)

```
spline --definition=examples/matrix.yaml --report=html
```

For the moment you cannot specify the output path and filename; it will be written to current working directory as *pipeline.html*.

22.2 Example

Spline - Pipeline Visualization

Matrix	Stage: Test1	Stage: Test2	Stage: Test3
name: one **duration**: 8.0 seconds	**status**: succeeded **duration**: 4.0 seconds	**status**: succeeded **duration**: 4.0 seconds	**status**: succeeded **duration**: 0.0 seconds
name: two **duration**: 6.0 seconds	**status**: succeeded **duration**: 3.0 seconds	**status**: failed **duration**: 3.0 seconds	no information
name: three **duration**: 2.0 seconds	**status**: succeeded **duration**: 1.0 seconds	**status**: succeeded **duration**: 1.0 seconds	**status**: failed **duration**: 0.0 seconds

Generated by the Spline tool, **version**: 1.7, **generated**: Sunday, 21. January 2018 - 03:12:22 PM

22.3 Multiprocessing

When running the matrixes in parallel then multiple processes are spawned. Using Python multiprocessing each process does send information via a queue to the collector (main process). The collector finally writes the *pipeline.yaml* on each update.

22.4 Refresh

The generated HTML does have a meta information that enforced refreshing of the page each 5 seconds allowing to see the progress of your pipelines.

CHAPTER 23

Development

23.1 Python Development

Many programming languages are providing essential language constructs and tools which help to write code with acceptable quality and giving you control to either keep or even to improve the quality constantly. **A decrease in quality should always fail the build**.

For me it turned that **tox** is a very useful tool to organize the Python build process (see **tox.ini**). Basically you define and reuse commands for different Python (virtual) environments. It's a wrapper for **virtualenv**. A good example is the definition for your tests:

```
[tool-test]
commands =
    coverage erase
    coverage run    --omit={toxinidir}/.tox/*,{toxinidir}/tests/* --branch -m␣
↪unittest discover -s {toxinidir}/tests -f -v
    coverage html   --title="Spline Code Coverage" --directory={toxinidir}/htmlcov
    coverage report --show-missing --fail-under={env:MIN_COVERAGE:95}
```

The calls:

```
tox -e test     # running tests and coverage
tox -e doctest  # running doctests only
```

The given example simply works with a standard Python installation and one additional tool named **coverage** (*pip install coverage*). The unittesting framework is capable of **discovering** all tests using following command:

```
python -m unittest discover -s {toxinidir}/tests -f -v
```

The parameter *-s* specifies the folder to start with, the parameter *-v* (verbose) does show each executed test method and the *-f* (failfast) stops immediately the tests on a failure. Using *coverage run* instead of *python* the whole runs also with code coverage. Additional parameters control what is included and/or excluded. In given case the *.tox* folder should be excluded since it contains all Python libraries which shouldn't be part of the coverage. In addition we would like to have more detailed information about the branch coverage.

The *coverage erase* ensure that results from a previous run do not influence the new coverage calculation. Finally you should be interested in two reports:

- **HTML report** - those one does show you the code in two colors: green=covered and red=not covered. It's easy then to see which tests you are missing.
- **Console report** - those one gives you quick and short summary. As last report also the limit is adjusted forcing the build to fail when the new coverage is less than the requested limit. A good orientation: try to have it greater or equal to 90%.

A special note on **MIN_COVERAGE**: Running the coverage for the spline project with spline itself you cannot run the Docker based tests because usually you can't run Docker inside Docker. Leaving out some tests the coverage will decrease and that's why the required coverage is decreased via the pipeline.yaml (but still above 90%).

How much coverage is needed? Maybe the most frustrating fact is that even you have 100% coverage the coverage is not necessarily complete. See following examples:

```
def square(n):
    return n*n
```

Let's say you write tests like `assert_that(square(2), equal_to(4))` you might think all is done but what happens if you call `square('2')`? You could argue that you wouldn't do that but as part of an calculation where the input has been read from a file or from stdin the usecase might be valid. Python doesn't enforce strict types.

```
def square(n):
    return int(n) * int(n)
```

Now with this function you can handle both (However you still miss floats). I don't say you should do that but the main focus here: also you have a coverage (line and branch) of 100% you might miss valid usecases. What we can say for sure: if you have less than 100% coverage you certainly will miss usecases.

Which test tool should be used? Very well known are **nosetests** and **pytest**. I leave it to you. For the spline project - trying to support many different Python versions - it turned out to run better without them.

What to do for static code analysis? For a long time **pep8**, **pep257**, **pylint**, **radon** and **flake8** are well known and often used tools. For pylint try to be as strict as possible:

- number of statements per method (or function)
- number of lines per file
- number of return statements
- number of parameters

There are more but reducing those numbers you can force yourself to care more on code design. Try to ensure that code complexity is as low as possible. Flake8 has an option to let it fail when the complexity of your code exceeds a definable limit (for spline: 6). Also try to keep line length acceptable; personally I wouldn't force to 80 but 110 is a value I felt comfortable with. Keep in mind that especially version diffs showing code side by side are influenced by this.

Some thrown warnings might annoy you sometimes but keeping the rules also mean to keep your code style consistent and that cannot be done without constant observation by tools. Before you commit your changes to the code repository run all tests and all analysis to be on the safe side. Here are the commands that can be used for individual checks:

```
tox -e pep8
tox -e pep257
tox -e pylint
tox -e flake8
tox -e radon
tox -e bandit
```

What about documentation? Tool documentation is one scenario and everybody who is using the tool should have reasonable documentation. You don't necessarily have to publish on read the docs but it should be easy to find via the main page of the project. You can read the spline documentation at read the docs as well as on the GitHub project. Another documentation is API documentation and especially interesting for developers intending to use the API. From what I have learned so far there are currently two good tools:

- **Sphinx**: The tool is not necessarily bound to code; you can just write markdown text or reStructuredText like this article. In addition there are extensions that allow embedding diagrams and code. The documentation of the tool itself is quite good.
- **epydoc**: this one is somehow similar to Doxygen and Javadoc; it seems that development has stopped (but that might have changed in the meantime). It's a very nice tool to get a good inside into the code.

I have used both. Please check the spline repository and also see how they are defined in the **tox.ini**.

```
tox -e sphinx  # generates read the doc HTML
tox -e apidoc  # generates API HTML with Sphinx
tox -e epydoc  # generates API HTML with epydoc
```

What about packaging? I decided to use wheel files. When installing the wheel file in your system all dependencies are installed as well. With **twine** (`pip install twine`) you can easily upload the package to **PyPI**.

```
tox -e package   # building the wheel file
```

I can advise only to be verbose in specifying the details for your package in your **setup.py** because there is much more than just uploading the code:

- of course you have to specify **name** and **version**
- the **long description** you should consider to read from a file and you can use reStructuredText.
- specifying author and a mail address
- specifying all package folders/paths
- you can specify **scripts** to be installed (like **spline**)
- you have to specify files that are not Python code (**package_data**)
- define the runtime dependencies (**install_requires**)
- The **url** can be any homepage for your component (tool or library)
- The **classifiers** is a standardized way to tell more about your component like **status** and which Python versions are supported, which platforms are supported and other informations like that.

How about testing Python versions you don't have on your machine? That has been one reason (there were others too) for writing the spline tool:

```
spline --matrix-tags=py27   # runs tox -e py27 inside Docker
spline --matrix-tags=py33   # runs tox -e py33 inside Docker
spline --matrix-tags=py34   # runs tox -e py34 inside Docker
spline --matrix-tags=py35   # runs tox -e py35 inside Docker
spline --matrix-tags=py36   # runs tox -e py36 inside Docker
spline --matrix-tags=pypy   # runs tox -e pypy inside Docker
spline --matrix-tags=pypy3  # runs tox -e pypy3 inside Docker
```

Because the different Python processes are running inside a well defined Docker container environment you are able to reproduce problems without affecting your own machine.

How about Travis CI? If you have completed all mentioned tasks the activating of Travis CI is easy. I have logged in with my GitHub account choosing the public repository and that it's. Now you require a **.travis.yml**. The file format

is quite simple; there's good documentation at Travis CI itself and you also can search for the file in the internet to find sufficient examples. You also can check the variant I have used in the spline project. The probably most interesting aspect for me was using of matrix builds. Two packages require attention:

- installation of *tox-travis* which ensures on a matrix build that tox understands which Python version has to be taken.
- installation of *coveralls* allows you to send coverage reports to the central service https://coveralls.io/. It also integrates as build check when doing pull requests being able to block a merge when coverage has decreased.

Finally here are **some links** you might find useful:

- https://tox.readthedocs.io/en/latest/
- http://coverage.readthedocs.io/en/latest/
- http://radon.readthedocs.io/en/latest/
- https://pylint.readthedocs.io/en/latest/
- https://pycodestyle.readthedocs.io/en/latest/
- http://pep257.readthedocs.io/en/latest/
- https://wiki.openstack.org/wiki/Security/Projects/Bandit
- https://docs.python.org/2/library/unittest.html
- https://docs.python.org/2/library/doctest.html
- http://pyhamcrest.readthedocs.io/en/latest/
- http://epydoc.sourceforge.net/
- http://www.sphinx-doc.org/en/stable/rest.html
- http://www.sphinx-doc.org/en/stable/ext/napoleon.html
- https://docs.travis-ci.com/user/languages/python/
- https://travis-ci.org/
- https://coveralls.io/

That's it. Please let me know when you miss details here. Also I'm interested in other tools that are useful for the Python build process that help to keep/improve the quality. Feel free to create a ticket (see issues on the GitHub page) with the details. Of course I will always update this article when I have new details.

CHAPTER 24

The spline-loc tool

24.1 Purpose

Helping to verify that the ratio between code and comments is at a level you can accept.

24.2 The usage

You can specify a path with *–path* (parameter is repeatable).

The threshold (ratio) is at 0.5 by default but you can specify with *–threshold* (or *-t*) to take another one you prefer. The threshold is for all files by default. At the moment Bash, Python, Java, Javascript, Typescript, Groovy and C++ are supported.

If one file has been found that is below given threshold the tool ends with exit code 1 (default).

```
$ spline-loc --path=spline
2018-08-11 11:04:34,790 - spline.tools.loc.application - Running with Python 2.7.13␣
↪(default, Nov 24 2017, 17:33:09) [GCC 6.3.0 20170516]
2018-08-11 11:04:34,798 - spline.tools.loc.application - Running on platform Linux-4.
↪9.0-6-amd64-x86_64-with-debian-9.4
2018-08-11 11:04:34,799 - spline.tools.loc.application - Current cpu count is 4
|-----|---|---|--------------------|------|
|Ratio|Loc|Com|File                |Type  |
|-----|---|---|--------------------|------|
|0.35 |162|57 |application.py      |Python|
|0.34 |77 |26 |tools/event.py      |Python|
|0.38 |89 |34 |tools/version.py    |Python|
|0.26 |213|56 |components/tasks.py |Python|
|0.36 |80 |29 |components/config.py|Python|
|-----|---|---|--------------------|------|
```

You can use the option *–show-all* (or *-s*) to show all files.

24.3 About loc, com and ratio

- **LOC** - lines of code without comments; empty lines included.
- **COM** - lines of comments; empty comment lines includes.
- **RATIO** - **COM / LOC** if **COM < LOC** otherwise 1.0.

Some notes:

- if you have as many comments as you have code the ratio is 1.0
- if you have one line comment for four lines code the ration is 0.25
- if you have comments only the ratio is 1.0
- if you have more comments than code the ratio is also 1.0

Basially I was interested in code that has not enough comments which focuses on ratios below 1.0. That's the idea.

24.4 About comments

- I do not check about empty lines.
- I do not check for sense ... if somebody writes 'bla bla bla' a code review should reject.
- I do not check tags against parameters because a) there are to many different styles and b) it would required to parse each language to know which parameters a function or method has.

24.5 Using average ratio only for valuation

The option *–average* does still report all files that have not enough documentation but the **spline-loc** tool (now) fails only when the average of all your ratios is smaller than your defined threshold:

```
$ spline-loc --path=spline --average
2018-08-14 05:44:03,157 - spline.tools.loc.application - Running with Python 2.7.13␣
↪(default, Nov 24 2017, 17:33:09) [GCC 6.3.0 20170516]
2018-08-14 05:44:03,221 - spline.tools.loc.application - Running on platform Linux-4.
↪9.0-6-amd64-x86_64-with-debian-9.4
2018-08-14 05:44:03,221 - spline.tools.loc.application - Current cpu count is 4
|-----|---|---|-----------------------|------|
|Ratio|Loc|Com|File                   |Type  |
|-----|---|---|-----------------------|------|
|0.35 |162|57 |application.py         |Python|
|0.34 |77 |26 |tools/event.py         |Python|
|0.38 |89 |34 |tools/version.py       |Python|
|0.26 |213|56 |components/tasks.py    |Python|
|0.36 |80 |29 |components/config.py   |Python|
|-----|---|---|-----------------------|------|
2018-08-14 05:44:03,240 - spline.tools.loc.application - average ratio is 0.72 for 34␣
↪files
```