Good practices

Data Intuitive Tuesday - January 26, 2021

- Unit testing in viash
  - Tests
- Run our component
- Run OCR on the png file
  - Platforms
  - Running the tests
  - Testing a namespace

In the previous sections, we wrote some software components and strung them together in Bash to form a simple pipeline. While these components work right now, they might soon break for any of the following reasons:

- While adding a feature to one of the components, a developer accidentally introduced a bug.
- A small change in the output of one component causes the other component to break.
- Breaking changes were introduced in the latest version of Python. The component's script needs to be updated or should be set to using an older version of Python.

If your component is actively developed, bugs will be introduced at some point. The question is, how long does it take for you to spot the errors? If it is located in infrequently used code, the bug might go unnoticed for months.

For this reason, you should add tests to software components which are being used in a lot of downstream analyses and/or whose results are of critical importance.

Unit testing in viash

We will introduce testing using the same `convert_plot` we used earlier to introduce the viash approach. We already covered the functionality of this component already in the previous sections. In this section, we

show how to add (unit) tests. Let see what the directory structure of the (updated) component looks like.

```
> tree src/civ6_save_renderer/convert_plot
src/civ6_save_renderer/convert_plot
├── config.vsh.yaml
├── script.sh
└── test
    └── run_test.sh

1 directory, 3 files
```

Just like in the viash primer (of the previous section) there is a viash config (`config.vsh.yaml`) and a script (`script.sh`). Nothing has changed to the script, but something was added to the viash config:

`src/civ6_save_renderer/convert_plot/config.vsh.yaml`:

```yaml
functionality:
  name: convert_plot
  namespace: civ6_save_renderer
  description: Convert a plot from pdf to png.
  version: "1.0"
  authors:
    - name: Robrecht Cannoodt
      email: rcannood@gmail.com
      roles: [maintainer, author]
      props: {github: rcannood, orcid: 0000-0003-3641-729X}
  arguments:
    - name: "--input"
      alternatives: [-i]
      type: file
      required: true
      default: "input.pdf"
      must_exist: true
      description: "A PDF input file."
    - name: "--output"
      alternatives: [-o]
      type: file
      required: true
      default: "output.png"
      direction: output
      description: "Output path."
  resources:
    - type: bash_script
      path: script.sh
  tests:
    - type: bash_script
      path: test/run_test.sh
    - path: https://www.w3.org/WAI/ER/tests/xhtml/testfiles/resources/pdf/dummy.pdf
platforms:
```

```
  - type: docker
    image: dpokidov/imagemagick
    setup:
      - type: apt
        packages: [ "tesseract-ocr" ]
  - type: nextflow
    image: dpokidov/imagemagick
  - type: native
```

The only differences with before are:

1. The addition of the `tests` under `functionality`.
2. An extra `apt` package to be installed for running the tests (see later).

Tests

Specifying the tests is not different from specifying the `resources` in the viash configuration. In this case, we have two resources: one is the script that contains the test code and one is a dummy PDF file that is fetched from the web during testing. We could also add a PDF file to the repository and point to that instead.

Let's take a look at the test script. #!/usr/bin/env bash

set -ex

Run our component

```
convert_plot
-i dummy.pdf
-o dummy.png
[[ ! -f dummy.png ]] && echo “No output generated!” && exit 1
```

Run OCR on the png file

tesseract dummy.png dummy-ocr

[[! `grep Dummy dummy-ocr.txt`]] && echo “Not the correct content” && exit 1

echo “»> Test finished successfully”

The test script itself defines two tests, namely: 1. A test to see if an output file is effectively created by our component 2. A test that extracts the text from the resulting `png` file in order to verify the content is still the same as the original.

In order to run the second step, we install a package `tesseract` that performs the OCR.

Platforms

The only difference with the `platforms` definition earlier is the installation of an additional package in the container.

Running the tests

In order to run the tests using the default platform (`docker` in our current example), we can simply run:

```
> viash test src/civ6_save_renderer/convert_plot/config.vsh.yaml
Running tests in temporary directory: '<...>/workspace/viash_temp/viash_test_convert_plot1100390243
============================================================================
+<...>/workspace/viash_temp/viash_test_convert_plot1100390243068873655/build_executable/convert_plo
> docker build -t civ6_save_renderer/convert_plot:1.0 <...>/workspace/viash_temp/viashsetupdocker-c
Sending build context to Docker daemon  17.41kB

Step 1/2 : FROM dpokidov/imagemagick
 ---> 0ce61e775be8
Step 2/2 : RUN apt-get update &&   apt-get install -y tesseract-ocr &&   rm -rf /var/lib/apt/lists/
 ---> Using cache
 ---> a92be5886a41
Successfully built a92be5886a41
Successfully tagged civ6_save_renderer/convert_plot:1.0
```

```
========================================================================
+<...>/workspace/viash_temp/viash_test_convert_plot1100390243068873655/test_run_test.sh/run_test.sh
+ convert_plot -i dummy.pdf -o dummy.png
convert: profile 'icc': 'RGB ': RGB color space not permitted on grayscale PNG `dummy.png' @ warnin
+ [[ ! -f dummy.png ]]
+ tesseract dummy.png dummy-ocr
Tesseract Open Source OCR Engine v4.0.0 with Leptonica
Warning: Invalid resolution 0 dpi. Using 70 instead.
Estimating resolution as 157
++ grep Dummy dummy-ocr.txt
+ [[ ! -n Dummy PDF file ]]
+ echo '>>> Test finished successfully'
>>> Test finished successfully
========================================================================
SUCCESS! All 1 out of 1 test scripts succeeded!
Cleaning up temporary directory
```

Let us break down what happens here:

1. viash creates a temporary directory (configurable via `$VIASH_TEMP`)
2. The setup of the appropriate platform is executed
3. The executable for the component is built in the temporary directory
4. The test script is run

If tests are successful, the temporary directory is removed (unless `--keep` is provided as an option to `viash test`).

This is a quick way to run a test on a component.

Testing a namespace

Similar to building a whole namespace, it is possible to test a whole namespace as well using the `viash ns test` command.

```
> viash ns test -p docker --parallel --tsv /tmp/report.tsv
          namespace        functionality             platform           test_name exit_code durat
  civ6_save_renderer         parse_header               docker               start
      markdown_tools         render_table               docker               start
  civ6_save_renderer        combine_plots               docker               start
  civ6_save_renderer            parse_map               docker               start
  civ6_save_renderer         convert_plot               docker               start
  civ6_save_renderer             plot_map               docker               start
  civ6_save_renderer         parse_header               docker     build_executable         0
  civ6_save_renderer         parse_header               docker               tests        -1
  civ6_save_renderer             plot_map               docker     build_executable         0
  civ6_save_renderer             plot_map               docker               tests        -1
  civ6_save_renderer         convert_plot               docker     build_executable         0
  civ6_save_renderer         convert_plot               docker          run_test.sh         0
      markdown_tools         render_table               docker     build_executable         0
      markdown_tools         render_table               docker               tests        -1
```

```
  civ6_save_renderer            parse_map              docker     build_executable         0
  civ6_save_renderer            parse_map              docker                tests        -1
  civ6_save_renderer        combine_plots              docker     build_executable         0
  civ6_save_renderer        combine_plots              docker           run_test.sh         0
```

With the `--parallel` option multiple tests are run in parallel (depending on your setup and the way Docker is configured).

The contents of (the optional) `report.tsv` contains a report of the test run:

namespace	functionality	platform	test_name	exit_code	duration	resul
civ6_save_renderer	parse_header	docker	build_executable	0	0	SUCC
civ6_save_renderer	parse_header	docker	tests	-1	0	MISSI
civ6_save_renderer	plot_map	docker	build_executable	0	0	SUCC
civ6_save_renderer	plot_map	docker	tests	-1	0	MISSI
civ6_save_renderer	convert_plot	docker	build_executable	0	0	SUCC
civ6_save_renderer	convert_plot	docker	run_test.sh	0	3	SUCC
markdown_tools	render_table	docker	build_executable	0	6	SUCC
markdown_tools	render_table	docker	tests	-1	0	MISSI
civ6_save_renderer	parse_map	docker	build_executable	0	6	SUCC
civ6_save_renderer	parse_map	docker	tests	-1	0	MISSI
civ6_save_renderer	combine_plots	docker	build_executable	0	6	SUCC
civ6_save_renderer	combine_plots	docker	run_test.sh	0	7	SUCC

For each component, you see the 2 steps from above: 1) build the executable and 2) run the actual test.

It should be noted that the tests are still running in their respective containers.