Good practices

Data Intuitive Tuesday - January 26, 2021

- Testing a component
  - Tests
  - Platforms
  - Running the tests
- Testing a namespace
- Continuous integration

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.

Testing a component

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`:

```
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_plot109163051
====================================================================
+<...>/workspace/viash_temp/viash_test_convert_plot10916305143432470291/build_executable/convert_pl
> docker build -t civ6_save_renderer/convert_plot:1.0 <...>/workspace/viash_temp/viashsetupdocker-
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_plot10916305143432470291/test_run_test.sh/run_test.s
+ 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 ]]
>>> Test finished successfully
+ echo '>>> 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
  civ6_save_renderer        combine_plots               docker                start
      markdown_tools         render_table               docker                start
  civ6_save_renderer             plot_map               docker                start
  civ6_save_renderer         convert_plot               docker                start
  civ6_save_renderer            parse_map               docker                start
  civ6_save_renderer         parse_header               docker     build_executable         0
  civ6_save_renderer             plot_map               docker     build_executable         0
  civ6_save_renderer             plot_map               docker                tests        -1
  civ6_save_renderer         parse_header               docker                tests        -1
  civ6_save_renderer            parse_map               docker     build_executable         0
  civ6_save_renderer            parse_map               docker                tests        -1
      markdown_tools         render_table               docker     build_executable         0
      markdown_tools         render_table               docker                tests        -1
  civ6_save_renderer         convert_plot               docker     build_executable         0
  civ6_save_renderer         convert_plot               docker          run_test.sh         0
  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). By specifying the `-p docker` flag, the tests will be run inside their respective containers – meaning that the test scripts and resources are automatically being copied in the container seamlessly.

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	plot_map	docker	build_executable	0	0	SUCC
civ6_save_renderer	plot_map	docker	tests	-1	0	MISSI
civ6_save_renderer	parse_header	docker	build_executable	0	0	SUCC
civ6_save_renderer	parse_header	docker	tests	-1	0	MISSI
civ6_save_renderer	parse_map	docker	build_executable	0	1	SUCC
civ6_save_renderer	parse_map	docker	tests	-1	0	MISSI
markdown_tools	render_table	docker	build_executable	0	2	SUCC
markdown_tools	render_table	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	5	SUCC
civ6_save_renderer	combine_plots	docker	build_executable	0	2	SUCC

namespace	functionality	platform	test_name	exit_code	duration	resul
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. You can also see that the developers of this codebase were lazy, as three out of five components are missing tests!

Continuous integration

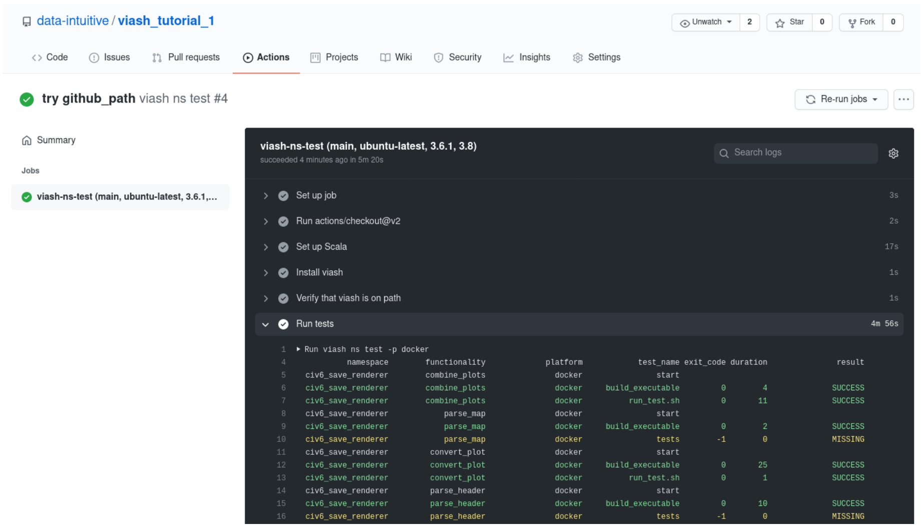
Continuous integration is a crucial DevOps practice for ensuring that any components work at all times. This is performed by adding all code to a code versioning repository (e.g. git) and configuring a CI service (e.g. GitHub Actions) to run all the tests on your code every time a change is made to the repository.

This repository is set up with GitHub Actions to run in this manner, and all it takes is yet another yaml file!

`.github/workflows/viash_ns_test.yml`:

```
name: viash ns test

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main, develop ]

jobs:
  viash-ns-test:
    runs-on: ${{ matrix.config.os }}
    if: "!contains(github.event.head_commit.message, 'ci skip')"

    strategy:
      fail-fast: false
      matrix:
        config:
        - {name: 'main', os: ubuntu-latest, r: '3.6.1', python: '3.8' }

    steps:
    - uses: actions/checkout@v2

    - name: Install viash
      run: |
        mkdir -p "$HOME/.local/bin"
        echo "$HOME/.local/bin" >> $GITHUB_PATH
        wget https://github.com/data-intuitive/viash/releases/download/v0.3.2/viash -qO "$HOME/.loc
        chmod +x "$HOME/.local/bin/viash"

    - name: Verify that viash is on path
      run: |
        viash -h

    - name: Run tests
      run: |
        viash ns test -p docker

    - name: Upload check results on fail
      if: failure()
      uses: actions/upload-artifact@master
      with:
        name: ${{ matrix.config.name }}_results
        path: check
```

Now, every time we commit something to the repository, the tests will be run:



data-intuitive / viash_tutorial_1
Unwatch 2
Star 0
Fork 0
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
try github_path viash ns test #4
Re-run jobs
Summary
Jobs
viash-ns-test (main, ubuntu-latest, 3.6.1,...
viash-ns-test (main, ubuntu-latest, 3.6.1, 3.8)
succeeded 4 minutes ago in 5m 20s
Search logs
Set up job 3s
Run actions/checkout@v2 2s
Set up Scala 17s
Install viash 1s
Verify that viash is on path 1s
Run tests 4m 56s

```
▸ Run viash ns test -p docker
          namespace        functionality          platform          test_name exit_code duration          result
  civ6_save_renderer       combine_plots            docker              start
  civ6_save_renderer       combine_plots            docker   build_executable         0        4         SUCCESS
  civ6_save_renderer       combine_plots            docker        run_test.sh         0       11         SUCCESS
  civ6_save_renderer           parse_map            docker              start
  civ6_save_renderer           parse_map            docker   build_executable         0        2         SUCCESS
  civ6_save_renderer           parse_map            docker              tests        -1        0         MISSING
  civ6_save_renderer        convert_plot            docker              start
  civ6_save_renderer        convert_plot            docker   build_executable         0       25         SUCCESS
  civ6_save_renderer        convert_plot            docker        run_test.sh         0        1         SUCCESS
  civ6_save_renderer        parse_header            docker              start
  civ6_save_renderer        parse_header            docker   build_executable         0       10         SUCCESS
  civ6_save_renderer        parse_header            docker              tests        -1        0         MISSING
```