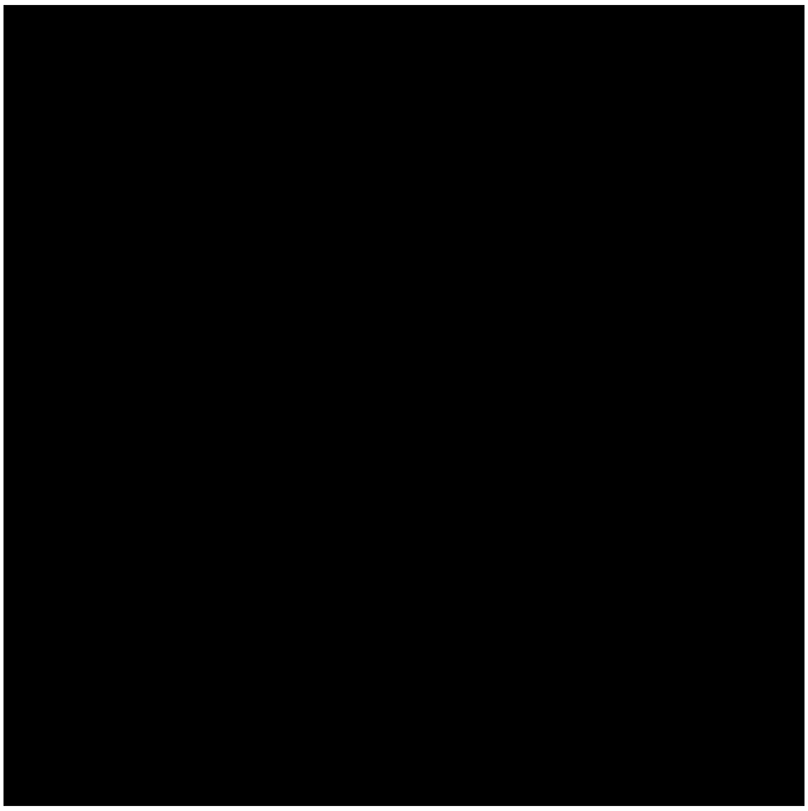
Doc as Test workshop

Table of Contents



1. Introduction

This workshop aims to experiment the 'Doc as Test' approach. You can find some information and resources about the concept at DocAsTest site.

In short, each test method generates a paragraph describing a behavior The aggregation of all those paragraphs makes a complete documentation of the product. As long as the generated text is the same, the test is passing. Otherwise, something has change and must be verify.

In this workshop we will work on a project that we will test and document. You can use any of your projects for this workshop. The most important thing is to be comfortable with your development environment. The constraints are that you need to be able to run tests on that project and you must be able to write a file to your disk from them. It also could be interesting (but not necessarily) to be able to push it in a public github repository to publish the final documentation and share it to other participants.

If you don't have a project, you can clone this repository: https://github.com/sfauvel/Parrot-Refactoring-Kata. This is a clone of the one on the Emily Bache repository.

This kata is available in many languages so you will probably find your favorite one. We have configured the java project with an advanced tooling. The python and javascript projects are also configured to start quickly.

2. Doc as code

In this section, we learn some basics about `Doc as code`: write a doc with a markup language, store it with source control, install tools to visualize it.

We propose to use Asciidoc to write documents. It's a markup language that makes it easy to write documents.

2.1. Install a viewer

Install an `asciidoc` viewer to view the rendering of the asciidoc files we are going to work on.. Prefer an installation on your IDE to see the result during the development.

▼ *Viewer list*

- Intellij/PyCharm: plugin
  - AsciiDoc plugin
- VSCode
  - AsciiDoc extension
- Web navigator
  - Firefox - Asciidoctor.js Live Preview
  - Chrome - Asciidoctor.js Live Preview
  - Edge - Asciidoctor.js Live Preview

2.2. Create a document

Create a `workshop.adoc` file. The `adoc` extension is used for `asciidoc` files.

Write an asciidoc file that describes the workshop you are in. Try to format it to be pleasant to read including:

- A title
- A little description about what you expect from this workshop highlighting key ideas.
- Links to useful resources.
- Table with information you want to give about people in your group.

You can found some basic syntaxes examples here or got to the syntax-quick-reference page of the asciidoc site.

2.3. Publish a site

It's not essential to publish an HTML documentation for this workshop. What we see in the `Asciidoc` viewer may be enough. But it is interesting to note the feeling of having an up-to-date support that is accessible by all continuously.

We will configure a github repository to publish our result.

▼ *How to publish site from Asciidoc with Github action*

The action we use creates a branch so, it needs to have write access on repository.

Go to `Settings > Actions > General > Workflow permissions` and allow workflows to write into the repository.

- [x] Workflows have read and write permissions in the repository for all scopes.
- [ ] Workflows have read permissions in the repository for the contents and packages scopes only.

We will add an action that creates HTML files from asciidoc files and commit them in a `gh-pages` branch.

If you cloned the https://github.com/sfauvel/Parrot-Refactoring-Kata repository, you find the `.github/workflows/publish_site.yaml` file that configure the action to publish your site. You just need to configure the path to your documentation folder. We suggest you put it in a "docs" directory in your test directory. You must have a `README.adoc` file in this folder. It'll become the `index.html` of you site.

If you use your own project, create the `publish_site.yaml` file with the following code and configure correctly the source dir :

Example 1. Publish action

```
name: GitHub Pages Publish

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

  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    # Includes the AsciiDoctor GitHub Pages Action to convert adoc files to html
and publish to gh-pages branch
    # from https://github.com/manoelcampos/asciidoctor-ghpages-action
    - name: asciidoctor-ghpages
      uses: manoelcampos/asciidoctor-ghpages-action@v2
      with:
        pdf_build: true
        asciidoctor_params: --attribute=htmlformat
        #asciidoctor_params: --attribute=nofooter --attribute=htmlformat
        # adoc_file_ext: .ascii # default is .adoc
        source_dir: pages/ # default is .
        # slides_build: true
        # pre_build: python pre_build.py
        # post_build:
```

Once the branch is created, we can publish its content. For that, got to: `Settings > Pages > Branch` and select `gh-pages`.

Your page is then visible at `https://[github account].github.io/[repository]/`.

Example: https://sfauvel.github.io/doc_as_test_workshop/

It will be updated on each commit on your `main` branch. It usually takes 1mn before you can see the result in your public site.

You also can look at the page Asciidoc to HTML which describes alternative ways of doing things.

3. Living documentation

After writing a static document, we will generate one to extract some information from the code.

For this workshop, you can use your own project or the Parrot refactoring kata. In the following, to guide you, we will suggest a path to follow with this kata. If you don’t use it, adapt the steps to your context and above all be free to explore other paths.

The first thing to do is make sure you can run the tests.

3.1. Extract info

Living documentation is documentation generated from code which is guaranteed to be up to date. Instead of creating this documentation from a program, we are going to use our tests like so many programs to generate the paragraphs of our documentation.

- Use a test to generate a file (with 'adoc' extension) containing information from the execution of the application code. If you use the Parrot kata with Java, Javascript, Typescript or Python, there is already a test that generate a file. The tests will fail but we will see how to make them passed in the next chapter.
- Write in this file the list of parrots as well as their default speed.
- Improve the writing to have a pleasant document to read.

4. Golden Master

Now, we have a documentation synchronized with the actual code behavior. The last step is to detect regression using `Golden Master` approach.

Now, you have a file which has captured the behavior of the application at a given time. If you run the test again, the result should be the same. Otherwise, that mean there is a regression since the last execution.

So, at the end of our test, we can compare the last generated file with the new version we produce. If they are identical, that’s fine. If it’s not, we just have to make our test fail.

We explain several ways to do it.

4.1. Prepare tooling

In this section, we explain several ways to set up minimal tools for this workshop. Choose one of them according to your context.

4.1.1. Approvals

You can use the Approvals library which is made for that and which is available in several languages.

In the kata Parrot, we have configured this library for the Java, Javascript and Python projects.

Otherwise, you need to configure it for your project and you have to specify you want to generate output files with `adoc` extension and not `txt` one. This can be a little tedious and it may be preferable to use one of the other two ways of doing things that we present next

`Approvals` use an `approved` file as expected reference. As long as your content is different from this file, a `received` file is generated to be able to compare the result with the expected one. If the `received` file is what you want, you just have to rename it or copy its content to the `approved` file to validate the new reference.

4.1.2. Use a script

You can just generate your files in a specific folder of your git repository. We provide a script `checkDocInFolder.sh` in `scripts` folder that check all files in the directory passes as parameter. If one file has changed, it's considered as a failing test.

To approve a file and make it the new reference, you just have to add it in staged files with command `git add [filename]`.

Usage example

```
bash scripts/checkDocInFolder.sh  ./Python/docs
```

The script is using the following git command to detect modified files:

```
git status -s --no-renames [FOLDER]
```

4.1.3. Create a method

If you don't want to integrate Approvals, you can write a simple method that does the job and is sufficient for our purposes.

A simple algorithm that can be integrated in a test framework could be the following one:

```
verify(filename, received_content):
  remove received_file_from(filename) if exists

  if approved_file_from(filename) not exists or approved_content != received_content:
    create received_file_from(filename)  with received_content
    fail
```

4.2. Validate existing behavior

We will create another chapter to see what happen to a parrot build with different parameters.

We want to understand what the impact of the voltage on Norwegian Blue speed.

Think first about the kind of description you want to see. What values you need to understand what happen ? Which input and output values ? The actions executed ? Then try to present it in a way that the behavior can quickly understood.

4.3. Validate new developments

> After experiment the approach on legacy code, we will try it for developing a new feature.

4.3.1. Specifications

We want to create a `area` where the `parrot` can travel. This `area` will contain the `parrot` position and `power sources` in some places. A `power source` has a value between 0 to 2.

We will develop a method to move the `parrot`. After 5 steps, its voltage decrease by 0.1. When a `parrot` passes over a `power source`, it takes all its energy.

▼ *Other exercises*

- Compute the time taken by the parrot during the travel.
- Create a document with only one general case and create a link to the document describing all behaviors.
- Create a json report of the board.
- Write an algorithm that move the parrot to consume all power sources.

4.3.2. Tips

- Focus on one behavior at a time.
- Creating a formatter that show the state of the system.
- You can wait to have a document that suits you before approving it (It’s not TDD).
- Several cases can be presented in the same chapter

5. Conclusion

You can send the url of your site on social networks (Twitter: `#DocAsTest`).

We will discuss your feelings after this experiment.

Thank you all for your participation.

6. Resources

This workshop

https://sfauvel.github.io/doc_as_test_workshop/

Parrot Kata

https://github.com/sfauvel/Parrot-Refactoring-Kata

Asciidoc

https://docs.asciidoctor.org/asciidoc/latest/

Asciidoc quick reference

https://docs.asciidoctor.org/asciidoc/latest/syntax-quick-reference/

Approvals

https://approvaltests.com/

DocAsTest

https://sfauvel.github.io/documentationtesting/
Human talk (video 10mn in French)
Bdx/IO (video 45mn in French)