

Living Documentation

Table of Contents

1. Introduction	1
2. Summary	2
3. Features	3
3.1. Cukedoctor Converter	3
3.1.1. Scenario: Convert features test output into documentation	3
3.2. Ordering	6
3.2.1. Scenario: Default ordering	6
3.2.2. Scenario: Custom ordering	7
3.3. Documentation introduction chapter	9
3.3.1. Scenario: Introduction chapter in classpath	9
3.4. Enrich features	13
3.4.1. Scenario: DocSting enrichment	13
3.4.2. Scenario: Comments enrichment	15

Chapter 1. Introduction

Cukedoctor is a **Living documentation** tool which integrates Cucumber and AsciiDoctor in order to convert your *BDD* tests results into an awesome documentation.

Here are some design principles:

- Living documentation should be readable and highlight your software features;
 - Most bdd tools generate reports and not a truly documentation.
- Cukedoctor **do not** introduce a new API that you need to learn, instead it operates on top of [cucumber json output](#) files;
 - In the 'worst case' to [enhance](#) your documentation you will need to know a bit of [asciidoc markup](#).

In the subsequent chapters you will see a documentation which is generated by the output of [Cukedoctor's BDD tests](#), a **real bdd living documentation**.

Chapter 2. Summary

Scenarios			Steps							Features: 4	
Passed	Failed	Total	Passed	Failed	Skipped	Pending	Undefined	Missing	Total	Duration	Status
Cukedoctor Converter											
1	0	1	3	0	0	0	0	0	3	02s 510ms	passed
Ordering											
2	0	2	6	0	0	0	0	0	6	061ms	passed
Documentation introduction chapter											
1	0	1	4	0	0	0	0	0	4	059ms	passed
Enrich features											
2	0	2	6	0	0	0	0	0	6	135ms	passed
Totals											
6	0	6	19	0	0	0	0	0	19	02s 765ms	

Chapter 3. Features

3.1. Cukedoctor Converter

In order to have awesome *living documentation*

As a bdd developer

I want to use **Cukedoctor** to convert my cucumber test results into **readable** living documentation

3.1.1. Scenario: Convert features test output into documentation

Given

The following two features: 🍌 (958ms)

Feature: Feature1

Scenario: Scenario feature 1

Given scenario step

Feature: Feature2

Scenario: Scenario feature 2

Given scenario step

When

I convert their json test output using cukedoctor converter 🍌 (01s 550ms)

To generate cucumber .json output files just execute your *BDD* tests with **json** formatter, example:



```
@RunWith(Cucumber.class)
@CucumberOptions(plugin = {"json:target/cucumber.json"})
)
```



plugin option replaced **format** option which was deprecated in newer cucumber versions.

Then

I should have awesome living documentation 🍌 (000ms)

Documentation

Summary

Scenarios			Steps							Features: 2	
Passe d	Faile d	Total	Passe d	Faile d	Skipp ed	Pendi ng	Undef ined	Missi ng	Total	Durat ion	Statu s
Feature1											
1	0	1	1	0	0	0	0	0	1	647ms	pas se d
Feature2											
1	0	1	1	0	0	0	0	0	1	000ms	pas se d
Totals											
2	0	2	2	0	0	0	0	0	2	647ms	

Features

Feature1

Scenario: Scenario feature 1

Given

scenario step 🍌 (647ms)

Feature2

Scenario: Scenario feature 2

Given

scenario step 🍌 (000ms)

3.2. Ordering

In order to have features ordered in living documentation
As a bdd developer
I want to control the order of features in my documentation

3.2.1. Scenario: Default ordering

Given

The following two features: 🍌 (000ms)

Feature: Feature1

Scenario: Scenario feature 1

Given scenario step

Feature: Feature2

Scenario: Scenario feature 2

Given scenario step

When

I convert them using default order 🍌 (030ms)

Then

Features should be ordered by name in resulting documentation 📱👍 (000ms)

Feature1

Scenario: Scenario feature 1

Given

scenario step 📱👍 (647ms)

Feature2

Scenario: Scenario feature 2

Given

scenario step 📱👍 (000ms)

3.2.2. Scenario: Custom ordering

Given

The following two features: 📊 (000ms)

```
#order: 2
```

```
Feature: Feature1
```

```
Scenario: Scenario feature 1
```

```
Given scenario step
```

```
#order: 1
```

```
Feature: Feature2
```

```
Scenario: Scenario feature 2
```

```
Given scenario step
```



Ordering is done using feature comment '**order:**'

When

I convert them using comment order 📊 (029ms)

Then

Features should be ordered respecting order comment 👍 (000ms)

Feature2

Scenario: Scenario feature 2

Given

scenario step 👍 (000ms)

Feature1

Scenario: Scenario feature 1

Given

scenario step 👍 (313ms)

3.3. Documentation introduction chapter

In order to have an introduction chapter in my documentation

As a bdd developer

I want to be able to provide an asciidoc based document which introduces my software

3.3.1. Scenario: Introduction chapter in classpath

Given

The following two features: 🍷 (001ms)

Feature: Feature1

Scenario: Scenario feature 1

Given scenario step

Feature: Feature2

Scenario: Scenario feature 2

Given scenario step

And

The following asciidoc document is on your application classpath 🍷 (057ms)

Introduction

Cukedoctor is a **Living documentation** tool which integrates Cucumber and Asciidoctor in order to convert your *BDD* tests results into an awesome documentation.

Here are some design principles:

Living documentation should be readable and highlight your software features;

Most bdd tools generate reports and not a truly documentation.

Cukedoctor **do not** introduce a new API that you need to learn, instead it operates on top of [cucumber json output](#) files;

In the 'worst case' to [enhance](#) your documentation you will need to know a bit of [asciidoc markup](#).



The introduction file must be named **intro-chapter.adoc** and can be in any package of your application,



By default Cukedoctor will look into application folders but you can make Cukedoctor look into external folder by setting the following system property:

```
System.setProperty("INTRO_CHAPTER_DIR", "/home/some/external/folder");
```

When

Bdd tests results are converted into documentation by Cukedoctor 🍷 (000ms)

Then

Resulting documentation should have the provided introduction chapter 🍷 (000ms)

Documentation

Introduction

Cukedoctor is a **Living documentation** tool which integrates Cucumber and AsciiDoctor in order to convert your *BDD* tests results into an awesome documentation.

Here are some design principles:

Living documentation should be readable and highlight your software features;

Most bdd tools generate reports and not a truly documentation.

Cukedoctor **do not** introduce a new API that you need to learn, instead it operates on top of [cucumber json output](#) files;

In the 'worst case' to [enhance](#) your documentation you will need to know a bit of [asciidoc markup](#).

Summary

Scenarios			Steps							Features: 2	
Passed	Failed	Total	Passed	Failed	Skipped	Pending	Undefined	Missing	Total	Duration	Status
Feature1											
1	0	1	1	0	0	0	0	0	1	647ms	passed
Feature2											
1	0	1	1	0	0	0	0	0	1	000ms	passed
Totals											
2	0	2	2	0	0	0	0	0	2	647ms	

Features

Feature1

Scenario: Scenario feature 1

Given

scenario step 🍷 (647ms)

Feature2

Scenario: Scenario feature 2

Given

scenario step 🍷 (000ms)

3.4. Enrich features

In order to have awesome *living documentation*
As a bdd developer
I want to render asciidoc markup inside my features

3.4.1. Scenario: DocSting enrichment

Asciiidoc markup can be used in feature **DocStrings**. To do so you need to enable it by using **cukector-dicrete** comment on the feature.

Given

The following two features: 🍌 (000ms)

Feature: Enrich feature

Scenario: Render source code

```
# cukedoctor-discrete
Given the following source code in docstrings
"""
[source, java]
-----
public int sum(int x, int y){
int result = x + y;
return result; (1)
}
-----
<1> We can have callouts in living documentation
"""
```

Scenario: Render table

```
# cukedoctor-discrete
Given the following table
"""
|==
| Cell in column 1, row 1 | Cell in column 2, row 1
| Cell in column 1, row 2 | Cell in column 2, row 2
| Cell in column 1, row 3 | Cell in column 2, row 3
|==
"""
```

When

I convert docstring enriched json output using cukedoctor converter 🍌 (084ms)

Then

Discrete class feature

Scenario: Render source code

Given

the following source code 👍 (267ms)

```
public int sum(int x, int y){  
    int result = x + y;  
    return result; ①  
}
```

① We can have callouts in living documentation>

Scenario: Render table

Given

the following table 👍 (000ms)

Cell in column 1, row 1	Cell in column 2, row 1
Cell in column 1, row 2	Cell in column 2, row 2
Cell in column 1, row 3	Cell in column 2, row 3

3.4.2. Scenario: Comments enrichment

Asciidoc markup can be used in feature comments. To do so you need to surround asciidoc markup by **curly brackets**;

Given

The following feature with asciidoc markup in comments: 🍌 (000ms)

Feature: Calculator

Scenario: Adding numbers

You can **asciidoc markup** in *_feature_* *#description#*.

NOTE: This is a very important feature!

*# {IMPORTANT: AsciiDoc markup inside **steps** must be surrounded by **curly brackets**.}*

Given I have numbers 1 and 2

*# {NOTE: Steps comments are placed **before** each steps so this comment is for the **WHEN** step.}*

When I sum the numbers

{ this is a list of itens inside a feature step}*

{ there is no multiline comment in gherkin}*

*# {** second level list item}*

Then I should have 3 as result

When

I convert enriched feature json output using cukedocter 🍌 (050ms)

Then

Calculator

Scenario: Adding numbers

You can use **asciidoc markup** in *feature* description.



This is a very important feature!

Given

I have numbers 1 and 2 👍 (114ms)



AsciiDoc markup inside **steps** must be surrounded by curly brackets.

When

I sum the numbers 👍 (000ms)



Steps comments are placed **before** each steps so this comment is for the **WHEN** step.

Then

I should have 3 as result 👍 (001ms)

this is a list of itens inside a feature step
there is no multiline comment in gherkin
second level list item