Living Documentation

Version 1.0.3-SNAPSHOT

Table of Contents

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												
0	1	1	2	1	0	0	0	0	3	02s 780ms	failed	
Ordering												
0	2	2	4	2	0	0	0	0	6	083ms	failed	
Enrich features												
1	1	2	5	1	0	0	0	0	6	132ms	failed	
Documentation introduction chapter												
0	1	1	3	1	0	0	0	0	4	042ms	failed	
Totals												
1	5	6	14	5	0	0	0	0	19	03s 038ms		

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: 👍 (918ms)

```
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 821ms)

> 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 👎 (039ms)

Documentation

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)

⛔ org.junit.ComparisonFailure: expected:<... Scenario feature 1 [

Given

> scenario step 👍 (647ms)

> === **Feature2**
>
> ==== Scenario: Scenario feature 2
>]

Given

> =====...> but was:<... Scenario feature 1
> [**

Given

> scenario step 👍 [small right]...

> === **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

> ==== 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 👍 (060ms)

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)

🛑 org.junit.ComparisonFailure: expected:<... Scenario feature 1
[

Given

scenario step 👍 (647ms)

=== **Feature2**

==== Scenario: Scenario feature 2
]

Given

=====...> but was:<... Scenario feature 1
[**

Given

scenario step 👍 [small right]...

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
```

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

When

I convert them using comment order 👍 (022ms)

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)

🛑   org.junit.ComparisonFailure: expected:<… Scenario feature 2
[

Given

scenario step 👍 (000ms)

3.2. Feature1

3.2.1. Scenario: Scenario feature 1

]

Given
=====…> but was:<… Scenario feature 2
[**

Given

scenario step 👍 [small right]…

3.3. Enrich features

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

3.3.1. Scenario: DocSting enrichment 👎

Asciidoc 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 👍 (057ms)

Then

DocString asciidoc output must be rendered in my documentation 👎 (000ms)

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

🛑 org.junit.ComparisonFailure: expected:<... 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 tabl...

3.3.2. 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 cukedoctor 👍 (073ms)

Then

Asciidoc markup on comments must be rendered in my documentation 👍 (000ms)

Calculator

Scenario: Adding numbers

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

> ℹ️ 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)

> ℹ️ 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

3.4. 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.

3.4.1. Scenario: Introduction chapter in classpath 👎

Given

The following two features: 👍 (000ms)

```
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 👍 (041ms)

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/f
> older");
> ```

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)

🛑 org.junit.ComparisonFailure: expected:<... Scenario feature 1
[

Given

scenario step 👍 (647ms)

> === **Feature2**
>
> ==== Scenario: Scenario feature 2
>]

Given

=====...> but was:<... Scenario feature 1
[**

Given

scenario step 👍 [small right]...