cucumber-documentation

Table of Contents

Features

After Hooks

After hooks can be used to clean up any state you’ve altered during your scenario, or to check the status of the scenario and act accordingly.

You can ask a scenario whether it has failed, for example.

Mind you, even if it hasn’t failed yet, you can still make the scenario fail if your After hook throws an error.

Background

Given

the standard step definitions 👍 (000ms)

Retreive the status of a scenario as a symbol

Given

a file named "features/support/debug_hook.rb" with: 👍 (000ms)

```
After do |scenario|
  puts scenario.status.inspect
end
```

And

a file named "features/result.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
```

When

I run `cucumber -f progress` 👍 (015ms)

Then

the output should contain ":passed" 👍 (000ms)

Retreive the status of a scenario as a symbol

Given

a file named "features/support/debug_hook.rb" with: 👍 (000ms)

```
After do |scenario|
  puts scenario.status.inspect
end
```

And

a file named "features/result.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step fails
```

When

I run `cucumber -f progress` 👍 (015ms)

Then

the output should contain ":failed" 👍 (000ms)

Retreive the status of a scenario as a symbol

Given

a file named "features/support/debug_hook.rb" with: 👍 (000ms)

```
After do |scenario|
  puts scenario.status.inspect
end
```

And

a file named "features/result.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step is pending
```

When

I run `cucumber -f progress` 👍 (013ms)

Then

the output should contain ":pending" 👍 (000ms)

Check the failed status of a scenario in a hook

Given

a file named "features/support/debug_hook.rb" with: 👍 (000ms)

```
After do |scenario|
  if scenario.failed?
    puts "eek"
  end
end
```

And

a file named "features/fail.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step fails
```

When

I run `cucumber -f progress` 👍 (012ms)

Then

the output should contain: 👍 (000ms)

```
eek
```

Make a scenario fail from an After hook

Given

a file named "features/support/bad_hook.rb" with: 👍 (000ms)

```
After do
  fail 'yikes'
end
```

And

a file named "features/pass.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
```

When

I run `cucumber -f pretty` 👍 (011ms)

Then

it should fail with: 👍 (000ms)

```
  Scenario:                 # features/pass.feature:2
    Given this step passes # features/step_definitions/steps.rb:1
      yikes (RuntimeError)
      ./features/support/bad_hook.rb:2:in `After'
```

After hooks are executed in reverse order of definition

Given

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
After do
  puts "First"
end

After do
  puts "Second"
end
```

And

a file named "features/pass.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
```

When

I run `cucumber -f progress` 👍 (007ms)

Then

the output should contain: 👍 (000ms)

```
Second

First
```

Alle bruker ikke UTF-8

Dette bør gå bra

Når

jeg drikker en "øl" 👍 (000ms)

Så

skal de andre si "skål" 👍 (000ms)

Around hooks

In order to support transactional scenarios for database libraries
that provide only a block syntax for transactions, Cucumber should
permit definition of Around hooks.

A single Around hook

tags: @spawn,@spawn

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Then /^the hook is called$/ do
  expect($hook_called).to be true
end
```

And

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
Around do |scenario, block|
  $hook_called = true
  block.call
end
```

And

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: Around hooks
  Scenario: using hook
    Then the hook is called
```

When

I run `cucumber features/f.feature` 👍 (605ms)

Then

it should pass with: 👍 (001ms)

```
Feature: Around hooks

  Scenario: using hook      # features/f.feature:2
    Then the hook is called # features/step_definitions/steps.rb:1

1 scenario (1 passed)
1 step (1 passed)
```

Multiple Around hooks

tags: @spawn,@spawn

Given

a file named "features/step_definitions/steps.rb" with: 👍 (001ms)

```
Then /^the hooks are called in the correct order$/ do
  expect($hooks_called).to eq ['A', 'B', 'C']
end
```

And

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
Around do |scenario, block|
  $hooks_called ||= []
  $hooks_called << 'A'
  block.call
end

Around do |scenario, block|
  $hooks_called ||= []
  $hooks_called << 'B'
  block.call
end

Around do |scenario, block|
  $hooks_called ||= []
  $hooks_called << 'C'
  block.call
end
```

And

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: Around hooks
  Scenario: using multiple hooks
    Then the hooks are called in the correct order
```

When

I run `cucumber features/f.feature` 👍 (607ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Around hooks

  Scenario: using multiple hooks                     # features/f.feature:2
    Then the hooks are called in the correct order #
features/step_definitions/steps.rb:1

1 scenario (1 passed)
1 step (1 passed)
```

Mixing Around, Before, and After hooks

tags: @spawn,@spawn

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Then /^the Around hook is called around Before and After hooks$/ do
  expect($hooks_called).to eq ['Around', 'Before']
end
```

And

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
Around do |scenario, block|
  $hooks_called ||= []
  $hooks_called << 'Around'
  block.call
  $hooks_called << 'Around'
  $hooks_called.should == ['Around', 'Before', 'After', 'Around'] #TODO:
Find out why this fails using the new rspec expect syntax.
end

Before do |scenario|
  $hooks_called ||= []
  $hooks_called << 'Before'
end

After do |scenario|
  $hooks_called ||= []
  $hooks_called << 'After'
  expect($hooks_called).to eq ['Around', 'Before', 'After']
end
```

And

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: Around hooks
  Scenario: Mixing Around, Before, and After hooks
    Then the Around hook is called around Before and After hooks
```

When

I run `cucumber features/f.feature` 👍 (607ms)

Then

it should pass with: 👍 (001ms)

```
Feature: Around hooks

  Scenario: Mixing Around, Before, and After hooks               #
features/f.feature:2
    Then the Around hook is called around Before and After hooks #
features/step_definitions/steps.rb:1

1 scenario (1 passed)
1 step (1 passed)
```

Around hooks with tags

tags: @spawn,@spawn

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Then /^the Around hooks with matching tags are called$/ do
  expect($hooks_called).to eq ['one', 'one or two']
end
```

And

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
Around('@one') do |scenario, block|
  $hooks_called ||= []
  $hooks_called << 'one'
  block.call
end

Around('@one,@two') do |scenario, block|
  $hooks_called ||= []
  $hooks_called << 'one or two'
  block.call
end

Around('@one', '@two') do |scenario, block|
  $hooks_called ||= []
  $hooks_called << 'one and two'
  block.call
end

Around('@two') do |scenario, block|
  $hooks_called ||= []
  $hooks_called << 'two'
  block.call
end
```

And

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: Around hooks
  @one
  Scenario: Around hooks with tags
    Then the Around hooks with matching tags are called
```

When

I run `cucumber -q -t @one features/f.feature` 👍 (708ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Around hooks

  @one
  Scenario: Around hooks with tags
    Then the Around hooks with matching tags are called

1 scenario (1 passed)
1 step (1 passed)
```

Around hooks with scenario outlines

tags: @spawn,@spawn

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Then /^the hook is called$/ do
  expect($hook_called).to be true
end
```

And

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
Around do |scenario, block|
  $hook_called = true
  block.call
end
```

And

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: Around hooks with scenario outlines
  Scenario Outline: using hook
    Then the hook is called

    Examples:
      | Number |
      | one    |
      | two    |
```

When

I run `cucumber features/f.feature` 👍 (607ms)

Then

it should pass with: 👍 (001ms)

```
Feature: Around hooks with scenario outlines

  Scenario Outline: using hook # features/f.feature:2
    Then the hook is called    # features/f.feature:3

    Examples:
      | Number |
      | one    |
      | two    |

2 scenarios (2 passed)
2 steps (2 passed)
```

Around Hooks and the Custom World

tags: @spawn,@spawn

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Then /^the world should be available in the hook$/ do
  $previous_world = self
  expect($hook_world).to eq(self)
end

Then /^what$/ do
  expect($hook_world).not_to eq($previous_world)
end
```

And

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
Around do |scenario, block|
  $hook_world = self
  block.call
end
```

And

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: Around hooks
  Scenario: using hook
    Then the world should be available in the hook

  Scenario: using the same hook
    Then what
```

When

I run `cucumber features/f.feature` 👍 (608ms)

Then

it should pass 👍 (000ms)

Background

Often you find that several scenarios in the same feature start with a common context.

Cucumber provides a mechanism for this, by providing a `Background` keyword where you can specify steps that should be run before each scenario in the feature. Typically these will be `Given` steps, but you can use any steps that you need to.

Hint: if you find that some of the scenarios don't fit the background, consider splitting them into a separate feature.

Background

Given

a file named "features/passing_background.feature" with: 👍 (000ms)

```
Feature: Passing background sample

  Background:
    Given '10' cukes

  Scenario: passing background
    Then I should have '10' cukes

  Scenario: another passing background
    Then I should have '10' cukes
```

And

a file named "features/scenario_outline_passing_background.feature" with: 👍 (000ms)

```
Feature: Passing background with scenario outlines sample

  Background:
    Given '10' cukes

  Scenario Outline: passing background
    Then I should have '<count>' cukes
    Examples:
      |count|
      | 10  |

  Scenario Outline: another passing background
    Then I should have '<count>' cukes
    Examples:
      |count|
      | 10  |
```

And

a file named "features/background_tagged_before_on_outline.feature" with: 👍 (000ms)

```
@background_tagged_before_on_outline
Feature: Background tagged Before on Outline

  Background:
    Given this step passes

  Scenario Outline: passing background
    Then I should have '<count>' cukes

    Examples:
      | count |
      | 888   |
```

And

a file named "features/failing_background.feature" with: 👍 (000ms)

```
Feature: Failing background sample

  Background:
    Given this step raises an error
    And '10' cukes

  Scenario: failing background
    Then I should have '10' cukes

  Scenario: another failing background
    Then I should have '10' cukes
```

And

a file named "features/scenario_outline_failing_background.feature" with: 👍 (000ms)

```
Feature: Failing background with scenario outlines sample

  Background:
    Given this step raises an error

  Scenario Outline: failing background
    Then I should have '<count>' cukes
    Examples:
      |count|
      | 10  |

  Scenario Outline: another failing background
    Then I should have '<count>' cukes
    Examples:
      |count|
      | 10  |
```

And

a file named "features/pending_background.feature" with: 👍 (000ms)

```
Feature: Pending background sample

  Background:
    Given this step is pending

  Scenario: pending background
    Then I should have '10' cukes

  Scenario: another pending background
    Then I should have '10' cukes
```

And

a file named "features/failing_background_after_success.feature" with: 👍 (000ms)

```
Feature: Failing background after previously successful background sample

  Background:
    Given this step passes
    And '10' global cukes

  Scenario: passing background
    Then I should have '10' global cukes

  Scenario: failing background
    Then I should have '10' global cukes
```

And

a file named "features/failing_background_after_success_outline.feature" with: 👍 (000ms)

```
Feature: Failing background after previously successful background sample

  Background:
    Given this step passes
    And '10' global cukes

  Scenario Outline: passing background
    Then I should have '<count>' global cukes

    Examples:
      | count |
      | 10    |

  Scenario Outline: failing background
    Then I should have '<count>' global cukes

    Examples:
      | count |
      | 10    |
```

And

a file named "features/multiline_args_background.feature" with: 👍 (000ms)

```
Feature: Passing background with multiline args

  Background:
    Given table
      |a|b|
      |c|d|
    And multiline string
      """
      I'm a cucumber and I'm okay.
      I sleep all night and I test all day
      """

  Scenario: passing background
    Then the table should be
      |a|b|
      |c|d|
    Then the multiline string should be
      """
      I'm a cucumber and I'm okay.
      I sleep all night and I test all day
      """

  Scenario: another passing background
    Then the table should be
      |a|b|
      |c|d|
    Then the multiline string should be
      """
      I'm a cucumber and I'm okay.
      I sleep all night and I test all day
      """
```

And

the standard step definitions 👍 (000ms)

And

a file named "features/step_definitions/cuke_steps.rb" with: 👍 (000ms)

```
Given /^'(.+)' cukes$/ do |cukes| x=1
  raise "We already have #{@cukes} cukes!" if @cukes
  @cukes = cukes
end

Given /^'(.+)' global cukes$/ do |cukes| x=1
  $scenario_runs ||= 0
  raise 'FAIL' if $scenario_runs >= 1
  $cukes = cukes
  $scenario_runs += 1
end

Then /^I should have '(.+)' global cukes$/ do |cukes| x=1
  expect($cukes).to eq cukes
end

Then /^I should have '(.+)' cukes$/ do |cukes| x=1
  expect(@cukes).to eq cukes
end

Before('@background_tagged_before_on_outline') do
  @cukes = '888'
end

After('@background_tagged_before_on_outline') do
  expect(@cukes).to eq '888'
end
```

run a specific scenario with a background

When

I run `cucumber -q features/passing_background.feature:9` 👍 (013ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: Passing background sample

  Background:
    Given '10' cukes

  Scenario: another passing background
    Then I should have '10' cukes

1 scenario (1 passed)
2 steps (2 passed)
```

run a feature with a background that passes

When

I run `cucumber -q features/passing_background.feature` 👍 (014ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: Passing background sample

  Background:
    Given '10' cukes

  Scenario: passing background
    Then I should have '10' cukes

  Scenario: another passing background
    Then I should have '10' cukes

2 scenarios (2 passed)
4 steps (4 passed)
```

run a feature with scenario outlines that has a background that passes

When

I run `cucumber -q features/scenario_outline_passing_background.feature` 👍 (012ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: Passing background with scenario outlines sample

  Background:
    Given '10' cukes

  Scenario Outline: passing background
    Then I should have '<count>' cukes

    Examples:
      | count |
      | 10    |

  Scenario Outline: another passing background
    Then I should have '<count>' cukes

    Examples:
      | count |
      | 10    |

2 scenarios (2 passed)
4 steps (4 passed)
```

run a feature with scenario outlines that has a background that passes

When

I run `cucumber -q features/background_tagged_before_on_outline.feature` 👍 (009ms)

Then

it should pass with exactly: 👍 (000ms)

```
@background_tagged_before_on_outline
Feature: Background tagged Before on Outline

  Background:
    Given this step passes

  Scenario Outline: passing background
    Then I should have '<count>' cukes

    Examples:
      | count |
      | 888   |

1 scenario (1 passed)
2 steps (2 passed)
```

run a feature with a background that fails

tags: @spawn

When

I run `cucumber -q features/failing_background.feature` 👍 (505ms)

Then

it should fail with exactly: 👍 (001ms)

```
Feature: Failing background sample

  Background:
    Given this step raises an error
      error (RuntimeError)
      ./features/step_definitions/steps.rb:2:in `/^this step raises an
error$/'
      features/failing_background.feature:4:in `Given this step raises an
error'
    And '10' cukes

  Scenario: failing background
    Then I should have '10' cukes

  Scenario: another failing background
    Then I should have '10' cukes

Failing Scenarios:
cucumber features/failing_background.feature:7
cucumber features/failing_background.feature:10

2 scenarios (2 failed)
6 steps (2 failed, 4 skipped)
```

run a feature with scenario outlines that has a background that fails

tags: @spawn

When

I run `cucumber -q features/scenario_outline_failing_background.feature` 👍 (605ms)

Then

it should fail with exactly: 👍 (001ms)

```
Feature: Failing background with scenario outlines sample

  Background:
    Given this step raises an error
      error (RuntimeError)
      ./features/step_definitions/steps.rb:2:in `/^this step raises an
error$/'
      features/scenario_outline_failing_background.feature:4:in `Given this
step raises an error'

  Scenario Outline: failing background
    Then I should have '<count>' cukes

    Examples:
      | count |
      | 10    |

  Scenario Outline: another failing background
    Then I should have '<count>' cukes

    Examples:
      | count |
      | 10    |

Failing Scenarios:
cucumber features/scenario_outline_failing_background.feature:10
cucumber features/scenario_outline_failing_background.feature:16

2 scenarios (2 failed)
4 steps (2 failed, 2 skipped)
```

run a feature with a background that is pending

When

I run `cucumber -q features/pending_background.feature` 👍 (024ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: Pending background sample

  Background:
    Given this step is pending
      TODO (Cucumber::Pending)
      ./features/step_definitions/steps.rb:3:in `/^this step is pending$/'
      features/pending_background.feature:4:in `Given this step is pending'

  Scenario: pending background
    Then I should have '10' cukes

  Scenario: another pending background
    Then I should have '10' cukes

2 scenarios (2 pending)
4 steps (2 skipped, 2 pending)
```

background passes with first scenario but fails with second

tags: @spawn

When

I run `cucumber -q features/failing_background_after_success.feature` 👍 (605ms)

Then

it should fail with exactly: 👍 (001ms)

```
Feature: Failing background after previously successful background sample

  Background:
    Given this step passes
    And '10' global cukes

  Scenario: passing background
    Then I should have '10' global cukes

  Scenario: failing background
    And '10' global cukes
      FAIL (RuntimeError)
      ./features/step_definitions/cuke_steps.rb:8:in `/^'(.+)' global
cukes$/'
      features/failing_background_after_success.feature:5:in `And '10'
global cukes'
    Then I should have '10' global cukes

Failing Scenarios:
cucumber features/failing_background_after_success.feature:10

2 scenarios (1 failed, 1 passed)
6 steps (1 failed, 1 skipped, 4 passed)
```

background passes with first outline scenario but fails with second

tags: @spawn

When

I run `cucumber -q features/failing_background_after_success_outline.feature` 👍 (605ms)

Then

it should fail with exactly: 👍 (001ms)

```
Feature: Failing background after previously successful background sample

  Background:
    Given this step passes
    And '10' global cukes

  Scenario Outline: passing background
    Then I should have '<count>' global cukes

    Examples:
      | count |
      | 10    |

  Scenario Outline: failing background
    Then I should have '<count>' global cukes

    Examples:
      | count |
      | 10    |
      FAIL (RuntimeError)
      ./features/step_definitions/cuke_steps.rb:8:in `/^'(.+)' global
cukes$/'
      features/failing_background_after_success_outline.feature:5:in `And
'10' global cukes'

Failing Scenarios:
cucumber features/failing_background_after_success_outline.feature:19

2 scenarios (1 failed, 1 passed)
6 steps (1 failed, 1 skipped, 4 passed)
```

background passes with first outline scenario but fails with second (--expand)

tags: @spawn

When

I run `cucumber -x -q features/failing_background_after_success_outline.feature` 👍 (606ms)

Then

it should fail with exactly: 👍 (000ms)

```
Feature: Failing background after previously successful background sample

  Background:
    Given this step passes
    And '10' global cukes

  Scenario Outline: passing background
    Then I should have '<count>' global cukes

    Examples:

      Scenario: | 10 |
        Then I should have '10' global cukes

  Scenario Outline: failing background
    Then I should have '<count>' global cukes

    Examples:

      Scenario: | 10 |
        And '10' global cukes
      FAIL (RuntimeError)
      ./features/step_definitions/cuke_steps.rb:8:in `/^'(.+)' global
cukes$/'
      features/failing_background_after_success_outline.feature:5:in `And
'10' global cukes'
        Then I should have '10' global cukes

Failing Scenarios:
cucumber features/failing_background_after_success_outline.feature:19

2 scenarios (1 failed, 1 passed)
6 steps (1 failed, 1 skipped, 4 passed)
```

background with multline args

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given /^table$/ do |table| x=1
  @table = table
end

Given /^multiline string$/ do |string| x=1
  @multiline = string
end

Then /^the table should be$/ do |table| x=1
  expect(@table.raw).to eq table.raw
end

Then /^the multiline string should be$/ do |string| x=1
  expect(@multiline).to eq string
end
```

When

I run `cucumber -q features/multiline_args_background.feature` 👍 (025ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: Passing background with multiline args

  Background:
    Given table
      | a | b |
      | c | d |
    And multiline string
      """
      I'm a cucumber and I'm okay.
      I sleep all night and I test all day
      """

  Scenario: passing background
    Then the table should be
      | a | b |
      | c | d |
    Then the multiline string should be
      """
      I'm a cucumber and I'm okay.
      I sleep all night and I test all day
      """

  Scenario: another passing background
    Then the table should be
      | a | b |
      | c | d |
    Then the multiline string should be
      """
      I'm a cucumber and I'm okay.
      I sleep all night and I test all day
      """

2 scenarios (2 passed)
8 steps (8 passed)
```

Before Hook

Examine names of scenario and feature

Given

a file named "features/foo.feature" with: 👍 (000ms)

```
Feature: Feature name

  Scenario: Scenario name
    Given a step
```

And

a file named "features/support/hook.rb" with: 👍 (000ms)

```
names = []
Before do |scenario|
  expect(scenario).to_not respond_to(:scenario_outline)
  names << scenario.feature.name.split("\n").first
  names << scenario.name.split("\n").first
  if(names.size == 2)
    raise "NAMES:\n" + names.join("\n") + "\n"
  end
end
```

When

I run `cucumber` 👍 (028ms)

Then

the output should contain: 👍 (000ms)

```
  NAMES:
  Feature name
  Scenario name
```

Examine names of scenario outline and feature

Given

a file named "features/foo.feature" with: 👍 (000ms)

```
Feature: Feature name

  Scenario Outline: Scenario Outline name
    Given a <placeholder>

    Examples: Examples Table name
      | <placeholder> |
      | step          |
```

And

a file named "features/support/hook.rb" with: 👍 (000ms)

```
names = []
Before do |scenario|
  names << scenario.scenario_outline.feature.name.split("\n").first
  names << scenario.scenario_outline.name.split("\n").first
  names << scenario.name.split("\n").first
  if(names.size == 3)
    raise "NAMES:\n" + names.join("\n") + "\n"
  end
end
```

When

I run `cucumber` 👍 (015ms)

Then

the output should contain: 👍 (000ms)

```
      NAMES:
      Feature name
      Scenario Outline name, Examples Table name (#1)
      Scenario Outline name, Examples Table name (#1)
```

Choosing the language from the feature file header

In order to simplify command line and settings in IDEs, Cucumber picks up the parser language from a `# language` comment at the beginning of any feature file. See the examples below for the exact syntax.

LOLCAT

Given

a file named "features/lolcat.feature" with: 👍 (000ms)

```
# language: en-lol
OH HAI: STUFFING
  B4: HUNGRY
    I CAN HAZ EMPTY BELLY
  MISHUN: CUKES
    DEN KTHXBAI
```

When

I run `cucumber -i features/lolcat.feature -q` 👍 (010ms)

Then

it should pass with: 👍 (000ms)

```
# language: en-lol
OH HAI: STUFFING

  B4: HUNGRY
    I CAN HAZ EMPTY BELLY

  MISHUN: CUKES
    DEN KTHXBAI

1 scenario (1 undefined)
2 steps (2 undefined)
```

Cucumber --work-in-progress switch

In order to ensure that feature scenarios do not pass until they are expected to
Developers should be able to run cucumber in a mode that
- will fail if any scenario passes completely
- will not fail otherwise

Background

A passing and a pending feature

tags: @spawn

Given

the standard step definitions 👍 (000ms)

And

a file named "features/wip.feature" with: 👍 (000ms)

```
Feature: WIP
  @failing
  Scenario: Failing
    Given this step raises an error

  @undefined
  Scenario: Undefined
    Given this step is undefined

  @pending
  Scenario: Pending
    Given this step is pending

  @passing
  Scenario: Passing
    Given this step passes
```

And

a file named "features/passing_outline.feature" with: 👍 (000ms)

```
Feature: Not WIP
  Scenario Outline: Passing
    Given this step <what>

    Examples:
      | what   |
      | passes |
```

Pass with Failing Scenarios

tags: @spawn,@spawn

When

I run `cucumber -q -w -t @failing features/wip.feature` 👍 (606ms)

Then

the stderr should not contain anything 👍 (000ms)

Then

it should pass with: 👍 (000ms)

```
Feature: WIP

  @failing
  Scenario: Failing
    Given this step raises an error
      error (RuntimeError)
      ./features/step_definitions/steps.rb:2:in `/^this step raises an
error$/'
      features/wip.feature:4:in `Given this step raises an error'

Failing Scenarios:
cucumber features/wip.feature:3

1 scenario (1 failed)
1 step (1 failed)
```

And

the output should contain: 👍 (000ms)

```
The --wip switch was used, so the failures were expected. All is good.
```

Pass with Undefined Scenarios

tags: @spawn,@spawn

When

I run `cucumber -q -w -t @undefined features/wip.feature` 👍 (608ms)

Then

it should pass with: 👍 (000ms)

```
Feature: WIP

  @undefined
  Scenario: Undefined
    Given this step is undefined

1 scenario (1 undefined)
1 step (1 undefined)
```

And

the output should contain: 👍 (000ms)

```
The --wip switch was used, so the failures were expected. All is good.
```

Pass with Undefined Scenarios

tags: @spawn,@spawn

When

I run `cucumber -q -w -t @pending features/wip.feature` 👍 (606ms)

Then

it should pass with: 👍 (001ms)

```
Feature: WIP

  @pending
  Scenario: Pending
    Given this step is pending
      TODO (Cucumber::Pending)
      ./features/step_definitions/steps.rb:3:in `/^this step is pending$/'
      features/wip.feature:12:in `Given this step is pending'

1 scenario (1 pending)
1 step (1 pending)
```

And

the output should contain: 👍 (000ms)

```
The --wip switch was used, so the failures were expected. All is good.
```

Fail with Passing Scenarios

tags: @spawn,@spawn

When

I run `cucumber -q -w -t @passing features/wip.feature` 👍 (607ms)

Then

it should fail with: 👍 (000ms)

```
Feature: WIP

  @passing
  Scenario: Passing
    Given this step passes

1 scenario (1 passed)
1 step (1 passed)
```

And

the output should contain: 👍 (000ms)

```
The --wip switch was used, so I didn't expect anything to pass. These
scenarios passed:
(::) passed scenarios (::)

features/wip.feature:15:in `Scenario: Passing'
```

Fail with Passing Scenario Outline

tags: @spawn,@spawn

When

I run `cucumber -q -w features/passing_outline.feature` 👍 (707ms)

Then

it should fail with: 👍 (001ms)

```
Feature: Not WIP

  Scenario Outline: Passing
    Given this step <what>

    Examples:
      | what   |
      | passes |

1 scenario (1 passed)
1 step (1 passed)
```

And

the output should contain: 👍 (000ms)

```
The --wip switch was used, so I didn't expect anything to pass. These
scenarios passed:
(::) passed scenarios (::)

features/passing_outline.feature:7:in `Scenario Outline: Passing, Examples
(#1)'
```

Custom Formatter

Background

Given

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: I'll use my own
  Scenario: Just print me
    Given this step passes
```

And

the standard step definitions 👍 (000ms)

Use the new API

Given

a file named "features/support/custom_formatter.rb" with: 👍 (000ms)

```
module MyCustom
  class Formatter
    def initialize(runtime, io, options)
      @io = io
    end

    def before_test_case(test_case)
      feature = test_case.source.first
      scenario = test_case.source.last
      @io.puts feature.short_name.upcase
      @io.puts "  #{scenario.name.upcase}"
    end
  end
end
```

When

I run `cucumber features/f.feature --format MyCustom::Formatter` 👍 (009ms)

Then

it should pass with exactly: 👍 (000ms)

```
I'LL USE MY OWN
  JUST PRINT ME
```

Use the legacy API

Given

a file named "features/support/custom_legacy_formatter.rb" with: 👍 (000ms)

```
module MyCustom
  class LegacyFormatter
    def initialize(runtime, io, options)
      @io = io
    end

    def before_feature(feature)
      @io.puts feature.short_name.upcase
    end

    def scenario_name(keyword, name, file_colon_line, source_indent)
      @io.puts "  #{name.upcase}"
    end
  end
end
```

When

I run `cucumber features/f.feature --format MyCustom::LegacyFormatter` 👍 (008ms)

Then

it should pass with exactly: 👍 (000ms)

```
I'LL USE MY OWN
  JUST PRINT ME
```

Use both

You can use a specific shim to opt-in to both APIs at once.

Given

a file named "features/support/custom_mixed_formatter.rb" with: 👍 (000ms)

```
module MyCustom
  class MixedFormatter

    def initialize(runtime, io, options)
      @io = io
    end

    def before_test_case(test_case)
      feature = test_case.source.first
      @io.puts feature.short_name.upcase
    end

    def scenario_name(keyword, name, file_colon_line, source_indent)
      @io.puts "  #{name.upcase}"
    end
  end
end
```

When

I run `cucumber features/f.feature --format MyCustom::MixedFormatter` 👍 (007ms)

Then

it should pass with exactly: 👍 (000ms)

```
I'LL USE MY OWN
  JUST PRINT ME
```

Custom filter

Add a custom filter via AfterConfiguration hook

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given my special step
```

And

a file named "features/support/my_filter.rb" with: 👍 (000ms)

```
require 'cucumber/core/filter'

MakeAnythingPass = Cucumber::Core::Filter.new do
  def test_case(test_case)
    activated_steps = test_case.test_steps.map do |test_step|
      test_step.with_action { }
    end
    test_case.with_steps(activated_steps).describe_to receiver
  end
end

AfterConfiguration do |config|
  config.filters << MakeAnythingPass.new
end
```

When

I run `cucumber --strict` 👍 (009ms)

Then

it should pass 👍 (000ms)

Debug formatter

In order to help you easily visualise the listener API, you can use the `debug` formatter that prints the calls to the listener as a feature is run.

Background

Given

the standard step definitions 👍 (000ms)

title

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
```

When

I run `cucumber -f debug` 👍 (007ms)

Then

the stderr should not contain anything 👍 (000ms)

Then

it should pass with: 👍 (000ms)

```
before_test_case
before_features
before_feature
before_tags
after_tags
feature_name
before_test_step
after_test_step
before_test_step
before_feature_element
before_tags
after_tags
scenario_name
before_steps
before_step
before_step_result
step_name
after_step_result
after_step
after_test_step
after_steps
after_feature_element
after_test_case
after_feature
after_features
done
```

Doc strings

If you need to specify information in a scenario that won't fit on a single line, you can use a DocString.

A DocString follows a step, and starts and ends with three double quotes, like this:

```
When I ask to reset my password
Then I should receive an email with:
  """
  Dear bozo,

  Please click this link to reset your password
  """
```

It's possible to annotate the DocString with the type of content it contains. This is used by formatting tools like http://relishapp.com which will render the contents of the DocString appropriately. You specify the content type after the triple quote, like this:

```
Given there is some Ruby code:
  """ruby
  puts "hello world"
  """
```

You can read the content type from the argument passed into your step definition, as shown in the example below.

Plain text Docstring

Given

a scenario with a step that looks like this: 👍 (000ms)

```
Given I have a lot to say:
 """
 One
 Two
 Three
 """
```

And

a step definition that looks like this: 👍 (000ms)

```
Given /say/ do |text|
  puts text
end
```

When

I run the feature with the progress formatter 👍 (010ms)

Then

the output should contain: 👍 (000ms)

```
One
Two
Three
```

DocString with interesting content type

Given

a scenario with a step that looks like this: 👍 (000ms)

```
Given I have some code for you:
 """ruby
 # hello
 """
```

And

a step definition that looks like this: 👍 (000ms)

```
Given /code/ do |text|
  puts text.content_type
end
```

When

I run the feature with the progress formatter 👍 (008ms)

Then

the output should contain: 👍 (000ms)

```
ruby
```

Dry Run

Dry run gives you a way to quickly scan your features without actually running them.

- Invokes formatters without executing the steps.
- This also omits the loading of your support/env.rb file if it exists.

With a failing step

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: test
  Scenario:
    Given this step fails
```

And

the standard step definitions 👍 (000ms)

When

I run `cucumber --dry-run` 👍 (020ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: test

  Scenario:               # features/test.feature:2
    Given this step fails # features/step_definitions/steps.rb:4

1 scenario (1 skipped)
1 step (1 skipped)
```

In strict mode

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: test
  Scenario:
    Given this step fails
```

And

the standard step definitions 👍 (000ms)

When

I run `cucumber --dry-run --strict` 👍 (013ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: test

  Scenario:               # features/test.feature:2
    Given this step fails # features/step_definitions/steps.rb:4

1 scenario (1 skipped)
1 step (1 skipped)
```

In strict mode with an undefined step

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: test
  Scenario:
    Given this step is undefined
```

When

I run `cucumber --dry-run --strict` 👍 (009ms)

Then

it should fail with: 👍 (000ms)

```
Feature: test

  Scenario:                      # features/test.feature:2
    Given this step is undefined # features/test.feature:3
      Undefined step: "this step is undefined" (Cucumber::Undefined)
      features/test.feature:3:in `Given this step is undefined'

1 scenario (1 undefined)
1 step (1 undefined)
```

ERB configuration

As a developer on server with multiple users
I want to be able to configure which port my wire server runs on
So that I can avoid port conflicts

Background

tags: @wire

Given

a file named "features/wired.feature" with: 👍 (000ms)

```
Feature: High strung
  Scenario: Wired
    Given we're all wired
```

ERB is used in the wire file which references an environment variable that is not set

tags: @wire,@wire

Given

a file named "features/step_definitions/server.wire" with: 👍 (000ms)

```
host: localhost
port: <%= ENV['PORT'] || 12345 %>
```

And

there is a wire server running on port 12345 which understands the following protocol: 👍 (002ms)

When

I run `cucumber --dry-run --no-snippets -f progress` 👍 (073ms)

Then

it should pass with: 👍 (000ms)

```
U

1 scenario (1 undefined)
1 step (1 undefined)
```

ERB is used in the wire file which references an environment variable

tags: @wire,@wire

Given

I have environment variable PORT set to "16816" 👍 (000ms)

And

a file named "features/step_definitions/server.wire" with: 👍 (000ms)

```
host: localhost
port: <%= ENV['PORT'] || 12345 %>
```

And

there is a wire server running on port 16816 which understands the following protocol: 👍 (002ms)

When

I run `cucumber --dry-run --no-snippets -f progress` 👍 (061ms)

Then

it should pass with: 👍 (001ms)

```
U

1 scenario (1 undefined)
1 step (1 undefined)
```

Exception in After Block

In order to use custom assertions at the end of each scenario
As a developer
I want exceptions raised in After blocks to be handled gracefully and reported by the formatters

Background

Given

the standard step definitions 👍 (000ms)

And

a file named "features/step_definitions/naughty_steps.rb" with: 👍 (000ms)

```
Given /^this step does something naughty$/ do x=1
  @naughty = true
end
```

And

a file named "features/support/env.rb" with: 👍 (000ms)

```
class NaughtyScenarioException < Exception; end
After do
  if @naughty
    raise NaughtyScenarioException.new("This scenario has been very very
naughty")
  end
end
```

Handle Exception in standard scenario step and carry on

tags: @spawn

Given

a file named "features/naughty_step_in_scenario.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario: Naughty Step
    Given this step does something naughty

  Scenario: Success
    Given this step passes
```

When

I run `cucumber features` 👍 (604ms)

Then

it should fail with: 👍 (000ms)

```
Feature: Sample

  Scenario: Naughty Step                    #
features/naughty_step_in_scenario.feature:3
    Given this step does something naughty #
features/step_definitions/naughty_steps.rb:1
      This scenario has been very very naughty (NaughtyScenarioException)
      ./features/support/env.rb:4:in `After'

  Scenario: Success        # features/naughty_step_in_scenario.feature:6
    Given this step passes # features/step_definitions/steps.rb:1

Failing Scenarios:
cucumber features/naughty_step_in_scenario.feature:3 # Scenario: Naughty
Step

2 scenarios (1 failed, 1 passed)
2 steps (2 passed)
```

Handle Exception in scenario outline table row and carry on

tags: @spawn

Given

a file named "features/naughty_step_in_scenario_outline.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario Outline: Naughty Step
    Given this step <Might Work>

    Examples:
    | Might Work             |
    | passes                 |
    | does something naughty |
    | passes                 |

  Scenario: Success
    Given this step passes
```

When

I run `cucumber features -q` 👍 (606ms)

Then

it should fail with: 👍 (000ms)

```
Feature: Sample

  Scenario Outline: Naughty Step
    Given this step <Might Work>

    Examples:
      | Might Work            |
      | passes                |
      | does something naughty |
      This scenario has been very very naughty (NaughtyScenarioException)
      ./features/support/env.rb:4:in `After'
      | passes                |

  Scenario: Success
    Given this step passes

Failing Scenarios:
cucumber features/naughty_step_in_scenario_outline.feature:9

4 scenarios (1 failed, 3 passed)
4 steps (4 passed)
```

Handle Exception using the progress format

Given

a file named "features/naughty_step_in_scenario.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario: Naughty Step
    Given this step does something naughty

  Scenario: Success
    Given this step passes
```

When

I run `cucumber features --format progress` 👍 (026ms)

Then

it should fail with: 👍 (000ms)

```
.F.

Failing Scenarios:
cucumber features/naughty_step_in_scenario.feature:3 # Scenario: Naughty
Step

2 scenarios (1 failed, 1 passed)
2 steps (2 passed)
```

Exception in AfterStep Block

In order to use custom assertions at the end of each step
As a developer
I want exceptions raised in AfterStep blocks to be handled gracefully and reported by the formatters

Background

Given

the standard step definitions 👍 (000ms)

And

a file named "features/step_definitions/naughty_steps.rb" with: 👍 (000ms)

```
Given /^this step does something naughty$/ do x=1
  @naughty = true
end
```

And

a file named "features/support/env.rb" with: 👍 (000ms)

```
class NaughtyStepException < Exception; end
AfterStep do
  if @naughty
    raise NaughtyStepException.new("This step has been very very naughty")
  end
end
```

Handle Exception in standard scenario step and carry on

Given

a file named "features/naughty_step_in_scenario.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario: Naughty Step
    Given this step does something naughty

  Scenario: Success
    Given this step passes
```

When

I run `cucumber features` 👍 (021ms)

Then

it should fail with: 👍 (000ms)

```
Feature: Sample

  Scenario: Naughty Step                     #
features/naughty_step_in_scenario.feature:3
    Given this step does something naughty #
features/step_definitions/naughty_steps.rb:1
      This step has been very very naughty (NaughtyStepException)
      ./features/support/env.rb:4:in `AfterStep'
      features/naughty_step_in_scenario.feature:4:in `Given this step does
something naughty'

  Scenario: Success        # features/naughty_step_in_scenario.feature:6
    Given this step passes # features/step_definitions/steps.rb:1

Failing Scenarios:
cucumber features/naughty_step_in_scenario.feature:3 # Scenario: Naughty
Step

2 scenarios (1 failed, 1 passed)
2 steps (2 passed)
```

Handle Exception in scenario outline table row and carry on

Given

a file named "features/naughty_step_in_scenario_outline.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario Outline: Naughty Step
    Given this step <Might Work>

    Examples:
    | Might Work             |
    | passes                 |
    | does something naughty |
    | passes                 |

  Scenario: Success
    Given this step passes
```

When

I run `cucumber features` 👍 (022ms)

Then

it should fail with: 👍 (000ms)

```
Feature: Sample

  Scenario Outline: Naughty Step #
features/naughty_step_in_scenario_outline.feature:3
    Given this step <Might Work> #
features/naughty_step_in_scenario_outline.feature:4

    Examples:
      | Might Work            |
      | passes                |
      | does something naughty |
      This step has been very very naughty (NaughtyStepException)
      ./features/support/env.rb:4:in `AfterStep'
      features/naughty_step_in_scenario_outline.feature:9:in `Given this
step does something naughty'
      features/naughty_step_in_scenario_outline.feature:4:in `Given this
step <Might Work>'
      | passes                |

  Scenario: Success        #
features/naughty_step_in_scenario_outline.feature:12
    Given this step passes # features/step_definitions/steps.rb:1

Failing Scenarios:
cucumber features/naughty_step_in_scenario_outline.feature:9 # Scenario
Outline: Naughty Step, Examples (#2)

4 scenarios (1 failed, 3 passed)
4 steps (4 passed)
```

Exception in Before Block

In order to know with confidence that my before blocks have run OK
As a developer
I want exceptions raised in Before blocks to be handled gracefully and reported by the formatters

Background

Given

the standard step definitions 👍 (000ms)

And

a file named "features/support/env.rb" with: 👍 (000ms)

```
class SomeSetupException < Exception; end
class BadStepException < Exception; end
Before do
  raise SomeSetupException.new("I cannot even start this scenario")
end
```

Handle Exception in standard scenario step and carry on

tags: @spawn

Given

a file named "features/naughty_step_in_scenario.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario: Run a good step
    Given this step passes
```

When

I run `cucumber features` 👍 (605ms)

Then

it should fail with: 👍 (001ms)

```
Feature: Sample

  Scenario: Run a good step # features/naughty_step_in_scenario.feature:3
  I cannot even start this scenario (SomeSetupException)
  ./features/support/env.rb:4:in `Before'
    Given this step passes  # features/step_definitions/steps.rb:1

Failing Scenarios:
cucumber features/naughty_step_in_scenario.feature:3 # Scenario: Run a good
step

1 scenario (1 failed)
1 step (1 skipped)
```

Handle Exception in Before hook for Scenario with Background

Given

a file named "features/naughty_step_in_before.feature" with: 👍 (000ms)

```
Feature: Sample

  Background:
    Given this step passes

  Scenario: Run a good step
    Given this step passes
```

When

I run `cucumber features` 👍 (023ms)

Then

it should fail with exactly: 👍 (000ms)

```
Feature: Sample

  Background:              # features/naughty_step_in_before.feature:3
  I cannot even start this scenario (SomeSetupException)
  ./features/support/env.rb:4:in `Before'
    Given this step passes # features/step_definitions/steps.rb:1

  Scenario: Run a good step # features/naughty_step_in_before.feature:6
    Given this step passes  # features/step_definitions/steps.rb:1

Failing Scenarios:
cucumber features/naughty_step_in_before.feature:6 # Scenario: Run a good
step

1 scenario (1 failed)
2 steps (2 skipped)
0m0.012s
```

Handle Exception using the progress format

Given

a file named "features/naughty_step_in_scenario.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario: Run a good step
    Given this step passes
```

When

I run `cucumber features --format progress` 👍 (016ms)

Then

it should fail with: 👍 (000ms)

```
F-

Failing Scenarios:
cucumber features/naughty_step_in_scenario.feature:3 # Scenario: Run a good
step

1 scenario (1 failed)
1 step (1 skipped)
```

Exceptions in Around Hooks

Around hooks are awkward beasts to handle internally.

Right now, if there's an error in your Around hook before you call `block.call`, we won't even print the steps for the scenario.

This is because that `block.call` invokes all the logic that would tell Cucumber's UI about the steps in your scenario. If we never reach that code, we'll never be told about them.

There's another scenario to consider, where the exception occurs after the steps have been run. How would we want to report in that case?

Exception before the test case is run

Given

the standard step definitions 👍 (000ms)

And

a file named "features/support/env.rb" with: 👍 (000ms)

```
Around do |scenario, block|
  fail "this should be reported"
  block.call
end
```

And

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
```

When

I run `cucumber -q` 👍 (010ms)

Then

it should fail with exactly: 👍 (000ms)

```
Feature:

  Scenario:
  this should be reported (RuntimeError)
  ./features/support/env.rb:2:in `Around'

Failing Scenarios:
cucumber features/test.feature:2

1 scenario (1 failed)
0 steps
```

Exception after the test case is run

Given

the standard step definitions 👍 (000ms)

And

a file named "features/support/env.rb" with: 👍 (000ms)

```
Around do |scenario, block|
  block.call
  fail "this should be reported"
end
```

And

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
```

When

I run `cucumber -q` 👍 (009ms)

Then

it should fail with exactly: 👍 (000ms)

```
Feature:

  Scenario:
    Given this step passes
      this should be reported (RuntimeError)
      ./features/support/env.rb:3:in `Around'

Failing Scenarios:
cucumber features/test.feature:2

1 scenario (1 failed)
1 step (1 passed)
```

Excluding ruby and feature files from runs

Developers are able to easily exclude files from cucumber runs
This is a nice feature to have in conjunction with profiles, so you can exclude certain environment files from certain runs.

exclude ruby files

Given

an empty file named "features/support/dont_require_me.rb" 👍 (000ms)

And

an empty file named "features/step_definitions/fooz.rb" 👍 (000ms)

And

an empty file named "features/step_definitions/foof.rb" 👍 (000ms)

And

an empty file named "features/step_definitions/foot.rb" 👍 (000ms)

And

an empty file named "features/support/require_me.rb" 👍 (000ms)

When

I run `cucumber features -q --verbose --exclude features/support/dont --exclude foo[zf]` 👍 (007ms)

Then

"features/support/require_me.rb" should be required 👍 (000ms)

And

"features/step_definitions/foot.rb" should be required 👍 (000ms)

And

"features/support/dont_require_me.rb" should not be required 👍 (000ms)

And

"features/step_definitions/foof.rb" should not be required 👍 (000ms)

And

"features/step_definitions/fooz.rb" should not be required 👍 (000ms)

[[Formatter-API:-Step-file-path-and-line-number-(Issue-#179), Formatter API: Step file path and line number (Issue #179)]]
=== **Formatter API: Step file path and line number (Issue #179)**

To all reporter to understand location of current executing step let’s fetch this information from step/step_invocation and pass to reporters

my own formatter

Given

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: I'll use my own
  because I'm worth it
  Scenario: just print step current line and feature file name
    Given step at line 4
    Given step at line 5
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given(/^step at line (.*)$/) {|line| }
```

And

a file named "features/support/jb/formatter.rb" with: 👍 (000ms)

```
module Jb
  class Formatter
    def initialize(runtime, io, options)
      @io = io
    end

    def before_step_result(keyword, step_match, multiline_arg, status,
exception, source_indent, background, file_colon_line)
      @io.puts "step result event: #{file_colon_line}"
    end

    def step_name(keyword, step_match, status, source_indent, background,
file_colon_line)
      @io.puts "step name event: #{file_colon_line}"
    end
  end
end
```

When

I run `cucumber features/f.feature --format Jb::Formatter` 👍 (006ms)

Then

it should pass with exactly: 👍 (000ms)

```
step result event: features/f.feature:4
step name event: features/f.feature:4
step result event: features/f.feature:5
step name event: features/f.feature:5
```

Getting started

To get started, just open a command prompt in an empty directory and run `cucumber`. You'll be prompted for what to do next.

Run Cucumber in an empty directory

tags: @spawn

Given

a directory without standard Cucumber project directory structure 👍 (000ms)

When

I run `cucumber` 👍 (605ms)

Then

it should fail with: 👍 (001ms)

```
No such file or directory - features. You can use `cucumber --init` to get
started.
```

Accidentally run Cucumber in a folder with Ruby files in it.

Given

a directory without standard Cucumber project directory structure 👍 (000ms)

And

a file named "should_not_load.rb" with: 👍 (000ms)

```
puts 'this will not be shown'
```

When

I run `cucumber` 👍 (007ms)

Then

the exit status should be 2 👍 (000ms)

And

the output should not contain: 👍 (000ms)

```
this will not be shown
```

HTML output formatter

Background

Given

the standard step definitions 👍 (000ms)

And

a file named "features/scenario_outline_with_undefined_steps.feature" with: 👍 (000ms)

```
Feature:

  Scenario Outline:
    Given this step is undefined

  Examples:
    |foo|
    |bar|
```

And

a file named "features/scenario_outline_with_pending_step.feature" with: 👍 (000ms)

```
Feature: Outline

  Scenario Outline: Will it blend?
    Given this step is pending
    And other step
    When I do something with <example>
    Then I should see something
    Examples:
      | example |
      | one     |
      | two     |
      | three   |
```

And

a file named "features/failing_background_step.feature" with: 👍 (000ms)

```
Feature: Feature with failing background step

  Background:
    Given this step fails

  Scenario:
    When I do something
    Then I should see something
```

an scenario outline, one undefined step, one random example, expand flag on

When

I run `cucumber features/scenario_outline_with_undefined_steps.feature --format html --expand ` 👍 (020ms)

Then

it should pass 👍 (003ms)

an scenario outline, one pending step

When

I run `cucumber features/scenario_outline_with_pending_step.feature --format html --expand` 👍 (025ms)

Then

it should pass 👍 (003ms)

And

the output should contain: 👍 (004ms)

```
makeYellow('scenario_1')
```

And

the output should not contain: 👍 (004ms)

```
makeRed('scenario_1')
```

an scenario outline, one pending step

When

I run `cucumber features/scenario_outline_with_pending_step.feature --format html ` 👍 (019ms)

Then

it should pass 👍 (002ms)

And

the output should contain: 👍 (003ms)

```
makeYellow('scenario_1')
```

And

the output should not contain: 👍 (004ms)

```
makeRed('scenario_1')
```

an scenario outline, one pending step

When

I run `cucumber features/scenario_outline_with_undefined_steps.feature --format html --expand` 👍 (007ms)

Then

it should pass 👍 (002ms)

And

the output should contain: 👍 (003ms)

```
makeYellow('scenario_1')
```

And

the output should not contain: 👍 (003ms)

```
makeRed('scenario_1')
```

an scenario outline, one pending step

When

I run `cucumber features/scenario_outline_with_undefined_steps.feature --format html ` 👍 (009ms)

Then

it should pass 👍 (002ms)

And

the output should contain: 👍 (003ms)

```
makeYellow('scenario_1')
```

And

the output should not contain: 👍 (003ms)

```
makeRed('scenario_1')
```

when using a profile the html shouldn’t include 'Using the default profile…'

And

a file named "cucumber.yml" with: 👍 (000ms)

```
  default: -r features
```

When

I run `cucumber features/scenario_outline_with_undefined_steps.feature --profile default --format html` 👍 (009ms)

Then

it should pass 👍 (002ms)

And

the output should not contain: 👍 (003ms)

```
Using the default profile...
```

a feature with a failing background step

When

I run `cucumber features/failing_background_step.feature --format html` 👍 (010ms)

Then

the output should not contain: 👍 (003ms)

```
makeRed('scenario_0')
```

And

the output should contain: 👍 (003ms)

```
makeRed('background_0')
```

Handle unexpected response

When the server sends us back a message we don’t understand, this is how Cucumber will behave.

Background

tags: @wire

Given

a file named "features/wired.feature" with: 👍 (000ms)

```
Feature: High strung
  Scenario: Wired
    Given we're all wired
```

And

a file named "features/step_definitions/some_remote_place.wire" with: 👍 (000ms)

```
host: localhost
port: 54321
```

Unexpected response

tags: @wire,@wire

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f pretty` 👍 (068ms)

Then

the output should contain: 👍 (000ms)

```
undefined method `handle_yikes'
```

Hooks execute in defined order

Background

tags: @spawn

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given /^background step$/ do; $EventOrder.push(:background_step) end
Given /^scenario step$/ do; $EventOrder.push(:scenario_step) end
```

And

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
$EventOrder = []
Around('@around') do |scenario,block|
  $EventOrder.push :around_begin
  block.call
  $EventOrder.push :around_end
end
Before('@before') do
  $EventOrder.push :before
end
After('@after') do |scenario|
  $EventOrder.push :after
end
at_exit {
  puts "Event order: #{$EventOrder.join(' ')}"
}
```

And

a file named "features/around_hook_covers_background.feature" with: 👍 (000ms)

```
@around
Feature: Around hooks cover background steps
  Background:
    Given background step
  Scenario:
    Given scenario step
```

And

a file named "features/all_hook_order.feature" with: 👍 (000ms)

```
@around
@before
@after
Feature: All hooks execute in expected order
  Background:
    Given background step
  Scenario:
    Given scenario step
```

Around hooks cover background steps

tags: @spawn,@spawn

When

I run `cucumber -o /dev/null features/around_hook_covers_background.feature` 👍 (606ms)

Then

the output should contain: 👍 (000ms)

```
Event order: around_begin background_step scenario_step around_end
```

All hooks execute in expected order

tags: @spawn,@spawn

When

I run `cucumber -o /dev/null features/all_hook_order.feature` 👍 (606ms)

Then

the output should contain: 👍 (000ms)

```
Event order: around_begin before background_step scenario_step after
around_end
```

Invoke message

Assuming a StepMatch was returned for a given step name, when it's time to invoke that step definition, Cucumber will send an invoke message.

The invoke message contains the ID of the step definition, as returned by the wire server in response to the the step_matches call, along with the arguments that were parsed from the step name during the same step_matches call.

The wire server will normally reply one of the following:

- `success`
- `fail`
- `pending` - optionally takes a message argument

This isn't quite the whole story: see also table_diffing.feature

Background

tags: @wire

Given

a file named "features/wired.feature" with: 👍 (000ms)

```
Feature: High strung
  Scenario: Wired
    Given we're all wired
```

And

a file named "features/step_definitions/some_remote_place.wire" with: 👍 (000ms)

```
host: localhost
port: 54321
```

Invoke a step definition which is pending

tags: @wire,@wire,@spawn

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (001ms)

When

I run `cucumber -f pretty -q` 👍 (806ms)

And

it should pass with: 👍 (001ms)

```
Feature: High strung

  Scenario: Wired
    Given we're all wired
      I'll do it later (Cucumber::Pending)
      features/wired.feature:3:in `Given we're all wired'

1 scenario (1 pending)
1 step (1 pending)
```

Invoke a step definition which passes

tags: @wire,@wire

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f progress` 👍 (140ms)

And

it should pass with: 👍 (000ms)

```
.

1 scenario (1 passed)
1 step (1 passed)
```

Invoke a step definition which fails

tags: @wire,@wire,@spawn

If an invoked step definition fails, it can return details of the exception
in the reply to invoke. This causes a Cucumber::WireSupport::WireException to be
raised.

Valid arguments are:

- `message` (mandatory)
- `exception`
- `backtrace`

See the specs for Cucumber::WireSupport::WireException for more details

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f progress` 👍 (808ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should fail with: 👍 (001ms)

```
F

(::) failed steps (::)

The wires are down (Some.Foreign.ExceptionType from localhost:54321)
features/wired.feature:3:in `Given we're all wired'

Failing Scenarios:
cucumber features/wired.feature:2 # Scenario: Wired

1 scenario (1 failed)
1 step (1 failed)
```

Invoke a step definition which takes string arguments (and passes)

tags: @wire,@wire

If the step definition at the end of the wire captures arguments, these are communicated back to Cucumber in the `step_matches` message.

Cucumber expects these StepArguments to be returned in the StepMatch. The keys have the following meanings:

- `val` - the value of the string captured for that argument from the step name passed in step_matches
- `pos` - the position within the step name that the argument was matched (used for formatter highlighting)

The argument values are then sent back by Cucumber in the `invoke` message.

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f progress` 👍 (141ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should pass with: 👍 (000ms)

```
.

1 scenario (1 passed)
1 step (1 passed)
```

Invoke a step definition which takes regular and table arguments (and passes)

tags: @wire,@wire

If the step has a multiline table argument, it will be passed with the invoke message as an array of array of strings.

In this scenario our step definition takes two arguments - one captures the "we’re" and the other takes the table.

Given

a file named "features/wired_on_tables.feature" with: 👍 (000ms)

```
Feature: High strung
  Scenario: Wired and more
    Given we're all:
      | wired |
      | high  |
      | happy |
```

And

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f progress features/wired_on_tables.feature` 👍 (139ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should pass with: 👍 (001ms)

```
.

1 scenario (1 passed)
1 step (1 passed)
```

Invoke a scenario outline step

tags: @wire,@wire

Given

a file named "features/wired_in_an_outline.feature" with: 👍 (000ms)

```
Feature:
  Scenario Outline:
    Given we're all <arg>

    Examples:
      | arg   |
      | wired |
```

And

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f progress features/wired_in_an_outline.feature` 👍 (146ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should pass with: 👍 (000ms)

```
.

1 scenario (1 passed)
1 step (1 passed)
```

And

the wire server should have received the following messages: 👍 (000ms)

JSON output formatter

In order to simplify processing of Cucumber features and results
Developers should be able to consume features as JSON

Background

Given

the standard step definitions 👍 (000ms)

And

a file named "features/one_passing_one_failing.feature" with: 👍 (000ms)

```
@a
Feature: One passing scenario, one failing scenario

  @b
  Scenario: Passing
    Given this step passes

  @c
  Scenario: Failing
    Given this step fails
```

And

a file named "features/step_definitions/json_steps.rb" with: 👍 (000ms)

```
Given /^I embed a screenshot/ do
  File.open("screenshot.png", "w") { |file| file << "foo" }
  embed "screenshot.png", "image/png"
end

Given /^I print from step definition/ do
  puts "from step definition"
end

Given /^I embed data directly/ do
  data = "YWJj"
  embed data, "mime-type;base64"
end
```

And

a file named "features/embed.feature" with: 👍 (000ms)

```
Feature: A screenshot feature

  Scenario:
    Given I embed a screenshot
```

And

a file named "features/outline.feature" with: 👍 (000ms)

```
Feature: An outline feature

  Scenario Outline: outline
    Given this step <status>

    Examples: examples1
      | status |
      | passes |
      | fails  |

    Examples: examples2
      | status |
      | passes |
```

And

a file named "features/print_from_step_definition.feature" with: 👍 (000ms)

```
Feature: A print from step definition feature

  Scenario:
    Given I print from step definition
    And I print from step definition
```

And

a file named "features/print_from_step_definition.feature" with: 👍 (000ms)

```
Feature: A print from step definition feature

  Scenario:
    Given I print from step definition
    And I print from step definition
```

And

a file named "features/embed_data_directly.feature" with: 👍 (000ms)

```
Feature: An embed data directly feature

  Scenario:
    Given I embed data directly

  Scenario Outline:
    Given I embed data directly

Examples:
| dummy |
|  1    |
|  2    |
```

And

a file named "features/out_scenario_out_scenario_outline.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
  Scenario Outline:
    Given this step <status>
    Examples:
    | status |
    | passes |
```

one feature, one passing scenario, one failing scenario

tags: @spawn

When

I run `cucumber --format json features/one_passing_one_failing.feature` 👍 (605ms)

Then

it should fail with JSON: 👍 (001ms)

```
[
  {
    "uri": "features/one_passing_one_failing.feature",
    "keyword": "Feature",
    "id": "one-passing-scenario,-one-failing-scenario",
    "name": "One passing scenario, one failing scenario",
    "line": 2,
    "description": "",
    "tags": [
      {
        "name": "@a",
        "line": 1
      }
    ],
    "elements": [
      {
        "keyword": "Scenario",
        "id": "one-passing-scenario,-one-failing-scenario;passing",
        "name": "Passing",
        "line": 5,
        "description": "",
        "tags": [
          {
            "name": "@a",
            "line": 1
          },
          {
            "name": "@b",
            "line": 4
          }
        ],
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step passes",
            "line": 6,
            "match": {
              "location": "features/step_definitions/steps.rb:1"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          }
        ]
      },
      {
        "keyword": "Scenario",
        "id": "one-passing-scenario,-one-failing-scenario;failing",
        "name": "Failing",
        "line": 9,
        "description": "",
        "tags": [
          {
            "name": "@a",
            "line": 1
          },
          {
            "name": "@c",
            "line": 8
          }
        ],
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step fails",
            "line": 10,
            "match": {
              "location": "features/step_definitions/steps.rb:4"
            },
            "result": {
              "status": "failed",
              "error_message": "
(RuntimeError)\n./features/step_definitions/steps.rb:4:in `/^this step
fails$/'\nfeatures/one_passing_one_failing.feature:10:in `Given this step
fails'",
              "duration": 1
            }
          }
        ]
      }
    ]
  }
]
```

one feature, one passing scenario, one failing scenario with prettyfied json

tags: @spawn

When

I run `cucumber --format json_pretty features/one_passing_one_failing.feature` 👍 (505ms)

Then

it should fail with JSON: 👍 (002ms)

```
[
  {
    "uri": "features/one_passing_one_failing.feature",
    "keyword": "Feature",
    "id": "one-passing-scenario,-one-failing-scenario",
    "name": "One passing scenario, one failing scenario",
    "line": 2,
    "description": "",
    "tags": [
      {
        "name": "@a",
        "line": 1
      }
    ],
    "elements": [
      {
        "keyword": "Scenario",
        "id": "one-passing-scenario,-one-failing-scenario;passing",
        "name": "Passing",
        "line": 5,
        "description": "",
        "tags": [
          {
            "name": "@a",
            "line": 1
          },
          {
            "name": "@b",
            "line": 4
          }
        ],
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step passes",
            "line": 6,
            "match": {
              "location": "features/step_definitions/steps.rb:1"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          }
        ]
      },
      {
        "keyword": "Scenario",
        "id": "one-passing-scenario,-one-failing-scenario;failing",
        "name": "Failing",
        "line": 9,
        "description": "",
        "tags": [
          {
            "name": "@a",
            "line": 1
          },
          {
            "name": "@c",
            "line": 8
          }
        ],
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step fails",
            "line": 10,
            "match": {
              "location": "features/step_definitions/steps.rb:4"
            },
            "result": {
              "status": "failed",
              "error_message": "
(RuntimeError)\n./features/step_definitions/steps.rb:4:in `/^this step
fails$/'\nfeatures/one_passing_one_failing.feature:10:in `Given this step
fails'",
              "duration": 1
            }
          }
        ]
      }
    ]
  }
]
```

DocString

tags: @spawn

Given

a file named "features/doc_string.feature" with: 👍 (000ms)

```
Feature: A DocString feature

  Scenario:
    Then I should fail with
      """
      a string
      """
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Then /I should fail with/ do |s|
  raise RuntimeError, s
end
```

When

I run `cucumber --format json features/doc_string.feature` 👍 (605ms)

Then

it should fail with JSON: 👍 (001ms)

```
[
  {
    "id": "a-docstring-feature",
    "uri": "features/doc_string.feature",
    "keyword": "Feature",
    "name": "A DocString feature",
    "line": 1,
    "description": "",
    "elements": [
      {
        "id": "a-docstring-feature;",
        "keyword": "Scenario",
        "name": "",
        "line": 3,
        "description": "",
        "type": "scenario",
        "steps": [
          {
            "keyword": "Then ",
            "name": "I should fail with",
            "line": 4,
            "doc_string": {
              "content_type": "",
              "value": "a string",
              "line": 5
            },
            "match": {
              "location": "features/step_definitions/steps.rb:1"
            },
            "result": {
              "status": "failed",
              "error_message": "a string
(RuntimeError)\n./features/step_definitions/steps.rb:2:in `/I should fail
with/'\nfeatures/doc_string.feature:4:in `Then I should fail with'",
              "duration": 1
            }
          }
        ]
      }
    ]
  }
]
```

embedding screenshot

tags: @spawn

When

I run `cucumber -b --format json features/embed.feature` 👍 (606ms)

Then

it should pass with JSON: 👍 (001ms)

```
[
  {
    "uri": "features/embed.feature",
    "id": "a-screenshot-feature",
    "keyword": "Feature",
    "name": "A screenshot feature",
    "line": 1,
    "description": "",
    "elements": [
      {
        "id": "a-screenshot-feature;",
        "keyword": "Scenario",
        "name": "",
        "line": 3,
        "description": "",
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "I embed a screenshot",
            "line": 4,
            "embeddings": [
              {
                "mime_type": "image/png",
                "data": "Zm9v"
              }
            ],
            "match": {
              "location": "features/step_definitions/json_steps.rb:1"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          }
        ]
      }
    ]
  }
]
```

scenario outline

tags: @spawn

When

I run `cucumber --format json features/outline.feature` 👍 (606ms)

Then

it should fail with JSON: 👍 (002ms)

```
[
  {
    "uri": "features/outline.feature",
    "id": "an-outline-feature",
    "keyword": "Feature",
    "name": "An outline feature",
    "line": 1,
    "description": "",
    "elements": [
      {
        "id": "an-outline-feature;outline;examples1;2",
        "keyword": "Scenario Outline",
        "name": "outline",
        "description": "",
        "line": 8,
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step passes",
            "line": 8,
            "match": {
              "location": "features/step_definitions/steps.rb:1"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          }
        ]
      },
      {
        "id": "an-outline-feature;outline;examples1;3",
        "keyword": "Scenario Outline",
        "name": "outline",
        "description": "",
        "line": 9,
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step fails",
            "line": 9,
            "match": {
              "location": "features/step_definitions/steps.rb:4"
            },
            "result": {
              "status": "failed",
              "error_message": "
(RuntimeError)\n./features/step_definitions/steps.rb:4:in `/^this step
fails$/'\nfeatures/outline.feature:9:in `Given this step
fails'\nfeatures/outline.feature:4:in `Given this step <status>'",
              "duration": 1
            }
          }
        ]
      },
      {
        "id": "an-outline-feature;outline;examples2;2",
        "keyword": "Scenario Outline",
        "name": "outline",
        "description": "",
        "line": 13,
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step passes",
            "line": 13,
            "match": {
              "location": "features/step_definitions/steps.rb:1"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          }
        ]
      }
    ]
  }
]
```

print from step definition

When

I run `cucumber --format json features/print_from_step_definition.feature` 👍 (013ms)

Then

it should pass with JSON: 👍 (000ms)

```
[
  {
    "uri": "features/print_from_step_definition.feature",
    "id": "a-print-from-step-definition-feature",
    "keyword": "Feature",
    "name": "A print from step definition feature",
    "line": 1,
    "description": "",
    "elements": [
      {
        "id": "a-print-from-step-definition-feature;",
        "keyword": "Scenario",
        "name": "",
        "line": 3,
        "description": "",
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "I print from step definition",
            "line": 4,
            "output": [
              "from step definition"
            ],
            "match": {
              "location": "features/step_definitions/json_steps.rb:6"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          },
          {
            "keyword": "And ",
            "name": "I print from step definition",
            "line": 5,
            "output": [
              "from step definition"
            ],
            "match": {
              "location": "features/step_definitions/json_steps.rb:6"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          }
        ]
      }
    ]
  }
]
```

scenario outline expanded

tags: @spawn

When

I run `cucumber --expand --format json features/outline.feature` 👍 (707ms)

Then

it should fail with JSON: 👍 (002ms)

```
[
  {
    "uri": "features/outline.feature",
    "id": "an-outline-feature",
    "keyword": "Feature",
    "name": "An outline feature",
    "line": 1,
    "description": "",
    "elements": [
      {
        "id": "an-outline-feature;outline;examples1;2",
        "keyword": "Scenario Outline",
        "name": "outline",
        "line": 8,
        "description": "",
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step passes",
            "line": 8,
            "match": {
              "location": "features/step_definitions/steps.rb:1"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          }
        ]
      },
      {
        "id": "an-outline-feature;outline;examples1;3",
        "keyword": "Scenario Outline",
        "name": "outline",
        "line": 9,
        "description": "",
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step fails",
            "line": 9,
            "match": {
              "location": "features/step_definitions/steps.rb:4"
            },
            "result": {
              "status": "failed",
              "error_message" : "
(RuntimeError)\n./features/step_definitions/steps.rb:4:in `/^this step
fails$/'\nfeatures/outline.feature:9:in `Given this step
fails'\nfeatures/outline.feature:4:in `Given this step <status>'",
"duration": 1
            }
          }
        ]
      },
      {
        "id": "an-outline-feature;outline;examples2;2",
        "keyword": "Scenario Outline",
        "name": "outline",
        "line": 13,
        "description": "",
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "this step passes",
            "line": 13,
            "match": {
              "location": "features/step_definitions/steps.rb:1"
            },
            "result": {
              "status": "passed",
              "duration": 1
            }
          }
        ]
      }
```

```
    ]
  }
]
```

embedding data directly

tags: @spawn

When

I run `cucumber -b --format json -x features/embed_data_directly.feature` 👍 (607ms)

Then

it should pass with JSON: 👍 (002ms)

```
[
  {
    "uri": "features/embed_data_directly.feature",
    "id": "an-embed-data-directly-feature",
    "keyword": "Feature",
    "name": "An embed data directly feature",
    "line": 1,
    "description": "",
    "elements": [
      {
        "id": "an-embed-data-directly-feature;",
        "keyword": "Scenario",
        "name": "",
        "line": 3,
        "description": "",
        "type": "scenario",
        "steps": [
          {
            "keyword": "Given ",
            "name": "I embed data directly",
            "line": 4,
            "embeddings": [
              {
  "mime_type": "mime-type",
  "data": "YWJj"
              }
            ],
            "match": {
              "location": "features/step_definitions/json_steps.rb:10"
            },
            "result": {
```

```
            "status": "passed",
            "duration": 1
          }
        }
      ]
    },
    {
      "keyword": "Scenario Outline",
      "name": "",
      "line": 11,
      "description": "",
      "id": "an-embed-data-directly-feature;;;2",
      "type": "scenario",
      "steps": [
        {
          "keyword": "Given ",
          "name": "I embed data directly",
          "line": 11,
          "embeddings": [
            {
              "mime_type": "mime-type",
              "data": "YWJj"
            }
          ],
          "match": {
            "location": "features/step_definitions/json_steps.rb:10"
          },
          "result": {
            "status": "passed",
            "duration": 1
          }
        }
      ]
    },
    {
      "keyword": "Scenario Outline",
      "name": "",
      "line": 12,
      "description": "",
      "id": "an-embed-data-directly-feature;;;3",
      "type": "scenario",
      "steps": [
        {
          "keyword": "Given ",
          "name": "I embed data directly",
          "line": 12,
          "embeddings": [
            {
              "mime_type": "mime-type",
              "data": "YWJj"
            }
          ],
          "match": {
            "location": "features/step_definitions/json_steps.rb:10"
          },
          "result": {
            "status": "passed",
            "duration": 1
          }
        }
      ]
    }
  ]
}
]
```

handle output from hooks

tags: @spawn

Given

a file named "features/step_definitions/output_steps.rb" with: 👍 (000ms)

```
Before do
  puts "Before hook 1"
  embed "src", "mime_type", "label"
end

Before do
  puts "Before hook 2"
  embed "src", "mime_type", "label"
end

AfterStep do
  puts "AfterStep hook 1"
  embed "src", "mime_type", "label"
end

AfterStep do
  puts "AfterStep hook 2"
  embed "src", "mime_type", "label"
end

After do
  puts "After hook 1"
  embed "src", "mime_type", "label"
end

After do
  puts "After hook 2"
  embed "src", "mime_type", "label"
end
```

When

I run `cucumber --format json features/out_scenario_out_scenario_outline.feature` 👍 (707ms)

Then

it should pass 👍 (001ms)

JUnit output formatter

In order for developers to create test reports with ant
Cucumber should be able to output JUnit xml files

Background

tags: @spawn

Given

the standard step definitions 👍 (000ms)

And

a file named "features/one_passing_one_failing.feature" with: 👍 (000ms)

```
Feature: One passing scenario, one failing scenario

  Scenario: Passing
    Given this step passes

  Scenario: Failing
    Given this step fails
```

And

a file named "features/some_subdirectory/one_passing_one_failing.feature" with: 👍 (000ms)

```
Feature: Subdirectory - One passing scenario, one failing scenario

  Scenario: Passing
    Given this step passes

  Scenario: Failing
    Given this step fails
```

And

a file named "features/pending.feature" with: 👍 (000ms)

```
Feature: Pending step

  Scenario: Pending
    Given this step is pending

  Scenario: Undefined
    Given this step is undefined
```

And

a file named "features/pending.feature" with: 👍 (000ms)

```
Feature: Pending step

  Scenario: Pending
    Given this step is pending

  Scenario: Undefined
    Given this step is undefined
```

And

a file named "features/scenario_outline.feature" with: 👍 (000ms)

```
Feature: Scenario outlines

  Scenario Outline: Using scenario outlines
    Given this step <type>

    Examples:
      | type         |
      | passes       |
      | fails        |
      | is pending   |
      | is undefined |
```

one feature, one passing scenario, one failing scenario

tags: @spawn,@spawn

When

I run `cucumber --format junit --out tmp/ features/one_passing_one_failing.feature` 👍 (706ms)

Then

it should fail with: 👍 (001ms)

And

the junit output file "tmp/TEST-features-one_passing_one_failing.xml" should contain: 👍 (000ms)

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite failures="1" errors="0" skipped="0" tests="2" time="0.05"
name="One passing scenario, one failing scenario">
<testcase classname="One passing scenario, one failing scenario"
name="Passing" time="0.05">
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="One passing scenario, one failing scenario"
name="Failing" time="0.05">
  <failure message="failed Failing" type="failed">
    <![CDATA[Scenario: Failing

Given this step fails

Message:
]]>
    <![CDATA[ (RuntimeError)
./features/step_definitions/steps.rb:4:in `/^this step fails$/'
features/one_passing_one_failing.feature:7:in `Given this step fails']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
</testsuite>
```

one feature in a subdirectory, one passing scenario, one failing scenario

tags: @spawn,@spawn

When

I run `cucumber --format junit --out tmp/features/some_subdirectory/one_passing_one_failing.feature --require features` 👍 (607ms)

Then

it should fail with: 👍 (000ms)

And

the junit output file "tmp/TEST-features-some_subdirectory-one_passing_one_failing.xml" should contain: 👍 (000ms)

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite failures="1" errors="0" skipped="0" tests="2" time="0.05"
name="Subdirectory - One passing scenario, one failing scenario">
<testcase classname="Subdirectory - One passing scenario, one failing
scenario" name="Passing" time="0.05">
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Subdirectory - One passing scenario, one failing
scenario" name="Failing" time="0.05">
  <failure message="failed Failing" type="failed">
    <![CDATA[Scenario: Failing

Given this step fails

Message:
]]>
    <![CDATA[ (RuntimeError)
./features/step_definitions/steps.rb:4:in `/^this step fails$/'
features/some_subdirectory/one_passing_one_failing.feature:7:in `Given this
step fails']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
</testsuite>
```

pending and undefined steps are reported as skipped

tags: @spawn,@spawn

When

I run `cucumber --format junit --out tmp/ features/pending.feature` 👍 (606ms)

Then

it should pass with: 👍 (001ms)

And

the junit output file "tmp/TEST-features-pending.xml" should contain: 👍 (000ms)

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite failures="0" errors="0" skipped="2" tests="2" time="0.05"
name="Pending step">
<testcase classname="Pending step" name="Pending" time="0.05">
  <skipped/>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Pending step" name="Undefined" time="0.05">
  <skipped/>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
</testsuite>
```

pending and undefined steps with strict option should fail

tags: @spawn,@spawn

When

I run `cucumber --format junit --out tmp/ features/pending.feature --strict` 👍 (706ms)

Then

```
it should fail with: 👍 (000ms)
```

And

the junit output file "tmp/TEST-features-pending.xml" should contain: 👍 (000ms)

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite failures="2" errors="0" skipped="0" tests="2" time="0.05"
name="Pending step">
<testcase classname="Pending step" name="Pending" time="0.05">
  <failure message="pending Pending" type="pending">
    <![CDATA[Scenario: Pending

Given this step is pending

Message:
]]>
    <![CDATA[TODO (Cucumber::Pending)
./features/step_definitions/steps.rb:3:in `/^this step is pending$/'
features/pending.feature:4:in `Given this step is pending']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Pending step" name="Undefined" time="0.05">
  <failure message="undefined Undefined" type="undefined">
    <![CDATA[Scenario: Undefined

Given this step is undefined

Message:
]]>
    <![CDATA[Undefined step: "this step is undefined"
(Cucumber::Core::Test::Result::Undefined)
features/pending.feature:7:in `Given this step is undefined']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
</testsuite>
```

run all features

tags: @spawn,@spawn

When

I run `cucumber --format junit --out tmp/ features` 👍 (707ms)

Then

it should fail with: 👍 (001ms)

And

a file named "tmp/TEST-features-one_passing_one_failing.xml" should exist 👍 (000ms)

And

a file named "tmp/TEST-features-pending.xml" should exist 👍 (000ms)

show correct error message if no --out is passed

tags: @spawn,@spawn

When

I run `cucumber --format junit features` 👍 (607ms)

Then

the stderr should not contain: 👍 (000ms)

```
can't convert .* into String \(TypeError\)
```

And

the stderr should contain: 👍 (000ms)

```
You *must* specify --out DIR for the junit formatter
```

strict mode, one feature, one scenario outline, four examples: one passing, one failing, one pending, one undefined

tags: @spawn,@spawn

When

I run `cucumber --strict --format junit --out tmp/ features/scenario_outline.feature` 👍 (707ms)

Then

it should fail with: 👍 (000ms)

And

the junit output file "tmp/TEST-features-scenario_outline.xml" should contain: 👍 (000ms)

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite failures="3" errors="0" skipped="0" tests="4" time="0.05"
name="Scenario outlines">
<testcase classname="Scenario outlines" name="Using scenario outlines
(outline example : | passes |)" time="0.05">
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Scenario outlines" name="Using scenario outlines
(outline example : | fails |)" time="0.05">
  <failure message="failed Using scenario outlines (outline example : |
fails |)" type="failed">
    <![CDATA[Scenario Outline: Using scenario outlines

Example row: | fails |

Message:
]]>
    <![CDATA[ (RuntimeError)
./features/step_definitions/steps.rb:4:in `/^this step fails$/'
features/scenario_outline.feature:9:in `Given this step fails'
features/scenario_outline.feature:4:in `Given this step <type>']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Scenario outlines" name="Using scenario outlines
(outline example : | is pending |)" time="0.05">
  <failure message="pending Using scenario outlines (outline example : | is
pending |)" type="pending">
    <![CDATA[Scenario Outline: Using scenario outlines

Example row: | is pending |

Message:
]]>
    <![CDATA[TODO (Cucumber::Pending)
./features/step_definitions/steps.rb:3:in `/^this step is pending$/'
features/scenario_outline.feature:10:in `Given this step is pending'
features/scenario_outline.feature:4:in `Given this step <type>']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Scenario outlines" name="Using scenario outlines
(outline example : | is undefined |)" time="0.05">
  <failure message="undefined Using scenario outlines (outline example : |
is undefined |)" type="undefined">
    <![CDATA[Scenario Outline: Using scenario outlines

Example row: | is undefined |

Message:
]]>
    <![CDATA[Undefined step: "this step is undefined"
(Cucumber::Core::Test::Result::Undefined)
features/scenario_outline.feature:11:in `Given this step is undefined'
features/scenario_outline.feature:4:in `Given this step <type>']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
</testsuite>
```

strict mode with --expand option, one feature, one scenario outline, four examples: one passing, one failing, one pending, one undefined

tags: @spawn,@spawn

When

I run `cucumber --strict --expand --format junit --out tmp/ features/scenario_outline.feature` 👍 (706ms)

Then

it should fail with exactly: 👍 (000ms)

And

the junit output file "tmp/TEST-features-scenario_outline.xml" should contain: 👍 (000ms)

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite failures="3" errors="0" skipped="0" tests="4" time="0.05"
name="Scenario outlines">
<testcase classname="Scenario outlines" name="Using scenario outlines
(outline example : | passes |)" time="0.05">
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Scenario outlines" name="Using scenario outlines
(outline example : | fails |)" time="0.05">
  <failure message="failed Using scenario outlines (outline example : |
fails |)" type="failed">
    <![CDATA[Scenario Outline: Using scenario outlines

Example row: | fails |

Message:
]]>
    <![CDATA[ (RuntimeError)
./features/step_definitions/steps.rb:4:in `/^this step fails$/'
features/scenario_outline.feature:9:in `Given this step fails'
features/scenario_outline.feature:4:in `Given this step <type>']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Scenario outlines" name="Using scenario outlines
(outline example : | is pending |)" time="0.05">
  <failure message="pending Using scenario outlines (outline example : | is
pending |)" type="pending">
    <![CDATA[Scenario Outline: Using scenario outlines

Example row: | is pending |

Message:
]]>
    <![CDATA[TODO (Cucumber::Pending)
./features/step_definitions/steps.rb:3:in `/^this step is pending$/'
features/scenario_outline.feature:10:in `Given this step is pending'
features/scenario_outline.feature:4:in `Given this step <type>']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
<testcase classname="Scenario outlines" name="Using scenario outlines
(outline example : | is undefined |)" time="0.05">
  <failure message="undefined Using scenario outlines (outline example : |
is undefined |)" type="undefined">
    <![CDATA[Scenario Outline: Using scenario outlines

Example row: | is undefined |

Message:
]]>
    <![CDATA[Undefined step: "this step is undefined"
(Cucumber::Core::Test::Result::Undefined)
features/scenario_outline.feature:11:in `Given this step is undefined'
features/scenario_outline.feature:4:in `Given this step <type>']]>
  </failure>
  <system-out>
    <![CDATA[]]>
  </system-out>
  <system-err>
    <![CDATA[]]>
  </system-err>
</testcase>
</testsuite>
```

Language help

It's possible to ask cucumber which keywords are used for any particular language by running:

```
cucumber --i18n <language code> help
```

This will print a table showing all the different words we use for that language, to allow you to easily write features in any language you choose.

Get help for Portuguese language

tags: @needs-many-fonts,@needs-many-fonts

When

I run `cucumber --i18n pt help` 👍 (007ms)

Then

it should pass with: 👍 (000ms)

```
      | feature          | "Funcionalidade", "Característica",
"Caracteristica"                                  |
      | background       | "Contexto", "Cenário de Fundo", "Cenario de
Fundo", "Fundo"                           |
      | scenario         | "Cenário", "Cenario"
|
      | scenario_outline | "Esquema do Cenário", "Esquema do Cenario",
"Delineação do Cenário", "Delineacao do Cenario" |
      | examples         | "Exemplos", "Cenários", "Cenarios"
|
      | given            | "* ", "Dado ", "Dada ", "Dados ", "Dadas "
|
      | when             | "* ", "Quando "
|
      | then             | "* ", "Então ", "Entao "
|
      | and              | "* ", "E "
|
      | but              | "* ", "Mas "
|
      | given (code)     | "Dado", "Dada", "Dados", "Dadas"
|
      | when (code)      | "Quando"
|
      | then (code)      | "Então", "Entao"
|
      | and (code)       | "E"
|
      | but (code)       | "Mas"
|
```

List languages

tags: @needs-many-fonts,@needs-many-fonts

When

I run `cucumber --i18n help` 👍 (006ms)

Then

cucumber lists all the supported languages 👍 (001ms)

List step defs as json

In order to build tools on top of Cucumber
As a tool developer
I want to be able to query a features directory for all the step definitions it contains

Background

tags: @spawn

Given

a directory named "features" 👍 (000ms)

Two Ruby step definitions, in the same file

tags: @spawn,@spawn

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given(/foo/i)  { }
Given(/b.r/xm) { }
```

When

I run the following Ruby code: 👍 (610ms)

```
require 'cucumber'
puts Cucumber::StepDefinitions.new.to_json
```

Then

it should pass with JSON: 👍 (000ms)

```
[
  {"source": "foo", "flags": "i"},
  {"source": "b.r", "flags": "mx"}
]
```

Non-default directory structure

tags: @spawn,@spawn

Given

a file named "my_weird/place/steps.rb" with: 👍 (000ms)

```
Given(/foo/)  { }
Given(/b.r/x) { }
```

When

I run the following Ruby code: 👍 (608ms)

```
require 'cucumber'
puts Cucumber::StepDefinitions.new(:autoload_code_paths =>
['my_weird']).to_json
```

Then

it should pass with JSON: 👍 (001ms)

```
[
  {"source": "foo", "flags": ""},
  {"source": "b.r", "flags": "x"}
]
```

Loading the steps users expect

As a User
In order to run features in subdirectories without having to pass extra options
I want cucumber to load all step files

Given

a file named "features/nesting/test.feature" with: 👍 (000ms)

```
Feature: Feature in Subdirectory
  Scenario: A step not in the subdirectory
    Given not found in subdirectory
```

And

a file named "features/step_definitions/steps_no_in_subdirectory.rb" with: 👍 (000ms)

```
Given(/^not found in subdirectory$/) { }
```

When

I run `cucumber -q features/nesting/test.feature` 👍 (006ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Feature in Subdirectory

  Scenario: A step not in the subdirectory
    Given not found in subdirectory

1 scenario (1 passed)
1 step (1 passed)
```

Nested Steps

Background

Given

a scenario with a step that looks like this: 👍 (000ms)

```
Given two turtles
```

And

a step definition that looks like this: 👍 (000ms)

```
Given /a turtle/ do
  puts "turtle!"
end
```

Use #steps to call several steps at once

Given

a step definition that looks like this: 👍 (000ms)

```
Given /two turtles/ do
  steps %{
    Given a turtle
    And a turtle
  }
end
```

When

I run the feature with the progress formatter 👍 (012ms)

Then

the output should contain: 👍 (000ms)

```
turtle!

turtle!
```

Use #step to call a single step

Given

a step definition that looks like this: 👍 (000ms)

```
Given /two turtles/ do
  step "a turtle"
  step "a turtle"
end
```

When

I run the feature with the progress formatter 👍 (007ms)

Then

the output should contain: 👍 (000ms)

```
turtle!

turtle!
```

Use #steps to call a table

Given

a step definition that looks like this: 👍 (000ms)

```
Given /turtles:/ do |table|
  table.hashes.each do |row|
    puts row[:name]
  end
end
```

And

a step definition that looks like this: 👍 (000ms)

```
Given /two turtles/ do
  steps %{
    Given turtles:
      | name      |
      | Sturm     |
      | Liouville |
  }
end
```

When

I run the feature with the progress formatter 👍 (008ms)

Then

the output should contain: 👍 (000ms)

```
Sturm

Liouville
```

Use #steps to call a multi-line string

Given

a step definition that looks like this: 👍 (000ms)

```
  Given /two turtles/ do
    steps %Q{
      Given turtles:
         """
         Sturm
         Liouville
         """
    }
  end
```

And

a step definition that looks like this: 👍 (000ms)

```
Given /turtles:/ do |string|
  puts string
end
```

When

I run the feature with the progress formatter 👍 (007ms)

Then

the output should contain: 👍 (000ms)

```
Sturm
Liouville
```

Backtrace doesn’t skip nested steps

tags: @spawn

Given

a step definition that looks like this: 👍 (000ms)

```
Given /two turtles/ do
  step "I have a couple turtles"
end

When(/I have a couple turtles/) { raise 'error' }
```

When

I run the feature with the progress formatter 👍 (607ms)

Then

it should fail with: 👍 (001ms)

```
error (RuntimeError)
./features/step_definitions/steps2.rb:5:in `/I have a couple turtles/'
./features/step_definitions/steps2.rb:2:in `/two turtles/'
features/test_feature_1.feature:3:in `Given two turtles'

Failing Scenarios:
cucumber features/test_feature_1.feature:2 # Scenario: Test Scenario 1

1 scenario (1 failed)
1 step (1 failed)
```

Undefined nested step

Given

a file named "features/call_undefined_step_from_step_def.feature" with: 👍 (000ms)

```
Feature: Calling undefined step

  Scenario: Call directly
    Given a step that calls an undefined step

  Scenario: Call via another
    Given a step that calls a step that calls an undefined step
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given /^a step that calls an undefined step$/ do
  step 'this does not exist'
end

Given /^a step that calls a step that calls an undefined step$/ do
  step 'a step that calls an undefined step'
end
```

When

I run `cucumber -q features/call_undefined_step_from_step_def.feature` 👍 (032ms)

Then

it should fail with exactly: 👍 (000ms)

```
Feature: Calling undefined step

  Scenario: Call directly
    Given a step that calls an undefined step
      Undefined dynamic step: "this does not exist"
(Cucumber::UndefinedDynamicStep)
      ./features/step_definitions/steps.rb:2:in `/^a step that calls an
undefined step$/'
      features/call_undefined_step_from_step_def.feature:4:in `Given a step
that calls an undefined step'

  Scenario: Call via another
    Given a step that calls a step that calls an undefined step
      Undefined dynamic step: "this does not exist"
(Cucumber::UndefinedDynamicStep)
      ./features/step_definitions/steps.rb:2:in `/^a step that calls an
undefined step$/'
      ./features/step_definitions/steps.rb:6:in `/^a step that calls a step
that calls an undefined step$/'
      features/call_undefined_step_from_step_def.feature:7:in `Given a step
that calls a step that calls an undefined step'

Failing Scenarios:
cucumber features/call_undefined_step_from_step_def.feature:3
cucumber features/call_undefined_step_from_step_def.feature:6

2 scenarios (2 failed)
2 steps (2 failed)
```

Nested Steps in I18n

Background

Given

a scenario with a step that looks like this in japanese: 👍 (000ms)

```
前提 two turtles
```

And

a step definition that looks like this: 👍 (000ms)

```
# -*- coding: utf-8 -*-
前提 /a turtle/ do
  puts "turtle!"
end
```

Use #steps to call several steps at once

Given

a step definition that looks like this: 👍 (000ms)

```
# -*- coding: utf-8 -*-
前提 /two turtles/ do
  steps %{
    前提 a turtle
    かつ a turtle
  }
end
```

When

I run the feature with the progress formatter 👍 (013ms)

Then

the output should contain: 👍 (000ms)

```
turtle!

turtle!
```

Nested Steps with either table or doc string

Background

Given

a scenario with a step that looks like this: 👍 (000ms)

```
Given two turtles
```

Use #step with table

Given

a step definition that looks like this: 👍 (000ms)

```
Given /turtles:/ do |table|
  table.hashes.each do |row|
    puts row[:name]
  end
end
```

And

a step definition that looks like this: 👍 (000ms)

```
Given /two turtles/ do
  step %{turtles:}, table(%{
  | name      |
  | Sturm     |
  | Liouville |
  })
end
```

When

I run the feature with the progress formatter 👍 (010ms)

Then

the output should contain: 👍 (000ms)

```
Sturm

Liouville
```

Use #step with docstring

Given

a step definition that looks like this: 👍 (000ms)

```
Given /two turtles/ do
  step %{turtles:}, "Sturm and Lioville"
end
```

And

a step definition that looks like this: 👍 (000ms)

```
Given /turtles:/ do |text|
  puts text
end
```

When

I run the feature with the progress formatter 👍 (009ms)

Then

the output should contain: 👍 (002ms)

```
Sturm and Lioville
```

Use #step with docstring and content-type

Given

a step definition that looks like this: 👍 (000ms)

```
Given /two turtles/ do
  step %{turtles:}, doc_string('Sturm and Lioville','math')
end
```

And

a step definition that looks like this: 👍 (000ms)

```
Given /turtles:/ do |text|
  puts text.content_type
end
```

When

I run the feature with the progress formatter 👍 (008ms)

Then

the output should contain: 👍 (000ms)

```
math
```

One line step definitions

Everybody knows you can do step definitions in Cucumber
but did you know you can do this?

Call a method in World directly from a step def

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
module Driver
  def do_action
    @done = true
  end

  def assert_done
    expect(@done).to be true
  end
end
World(Driver)

When /I do the action/, :do_action
Then /The action should be done/, :assert_done
```

And

a file named "features/action.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    When I do the action
    Then the action should be done
```

When

I run `cucumber` 👍 (008ms)

Then

it should pass 👍 (000ms)

Call a method on an actor in the World directly from a step def

Given

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
class Thing
  def do_action
    @done = true
  end

  def assert_done
    expect(@done).to be true
  end
end

module Driver
  def thing
    @thing ||= Thing.new
  end
end
World(Driver)

When /I do the action to the thing/, :do_action, :on => lambda { thing }
Then /The thing should be done/, :assert_done, :on => lambda { thing }
```

And

a file named "features/action.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    When I do the action to the thing
    Then the thing should be done
```

When

I run `cucumber` 👍 (008ms)

Then

it should pass 👍 (000ms)

[[Post-Configuration-Hook-[#423], Post Configuration Hook [#423]]]
=== **Post Configuration Hook [#423]**

In order to extend Cucumber
As a developer
I want to manipulate the Cucumber configuration after it has been created

Using options directly gets a deprecation warning

tags: @spawn,@wip-jruby

Given

a file named "features/support/env.rb" with: 👍 (000ms)

```
AfterConfiguration do |config|
  config.options[:blah]
end
```

When

I run `cucumber features` 👍 (605ms)

Then

the stderr should contain: 👍 (000ms)

```
Deprecated
```

Changing the output format

Given

a file named "features/support/env.rb" with: 👍 (000ms)

```
AfterConfiguration do |config|
  config.formats << ['html', config.out_stream]
end
```

When

I run `cucumber features` 👍 (016ms)

Then

the stderr should not contain anything 👍 (000ms)

And

the output should contain: 👍 (008ms)

```
html
```

feature directories read from configuration

Given

a file named "features/support/env.rb" with: 👍 (000ms)

```
AfterConfiguration do |config|
  config.out_stream << "AfterConfiguration hook read feature directories:
#{config.feature_dirs.join(', ')}"
end
```

When

I run `cucumber features` 👍 (006ms)

Then

the stderr should not contain anything 👍 (000ms)

And

the output should contain: 👍 (000ms)

```
AfterConfiguration hook read feature directories: features
```

Pretty formatter - Printing messages

When you want to print to Cucumber’s output, just call `puts` from a step definition. Cucumber will grab the output and print it via the formatter that you’re using.

Your message will be printed out after the step has run.

Background

Given

the standard step definitions 👍 (000ms)

And

a file named "features/step_definitions/puts_steps.rb" with: 👍 (000ms)

```
Given /^I use puts with text "(.*)"$/ do |ann|
  puts(ann)
end

Given /^I use multiple putss$/ do
  puts("Multiple")
  puts("Announce","Me")
end

Given /^I use message (.+) in line (.+) (?:with result (.+))$/ do |ann,
line, result|
  puts("Last message") if line == "3"
  puts("Line: #{line}: #{ann}")
  fail if result =~ /fail/i
end

Given /^I use puts and step fails$/ do
  puts("Announce with fail")
  fail
end

Given /^I puts the world$/ do
  puts(self)
end
```

And

a file named "features/f.feature" with: 👍 (000ms)

```
Feature:

  Scenario:
    Given I use puts with text "Ann"
    And this step passes

  Scenario:
    Given I use multiple putss
    And this step passes

  Scenario Outline:
    Given I use message <ann> in line <line>

    Examples:
      | line | ann   |
      | 1    | anno1 |
      | 2    | anno2 |
      | 3    | anno3 |

  Scenario:
    Given I use puts and step fails
    And this step passes

  Scenario Outline:
    Given I use message <ann> in line <line> with result <result>

    Examples:
      | line | ann   | result |
      | 1    | anno1 | fail   |
      | 2    | anno2 | pass   |
```

And

a file named "features/puts_world.feature" with: 👍 (000ms)

```
Feature: puts_world
  Scenario: puts_world
    Given I puts the world
```

Delayed messages feature

tags: @spawn

When

I run `cucumber --quiet --format pretty features/f.feature` 👍 (505ms)

Then

the stderr should not contain anything 👍 (000ms)

And

the output should contain: 👍 (000ms)

```
Feature:

  Scenario:
    Given I use puts with text "Ann"
      Ann
    And this step passes

  Scenario:
    Given I use multiple putss
      Multiple
      Announce
      Me
    And this step passes

  Scenario Outline:
    Given I use message <ann> in line <line>

    Examples:
      | line | ann   |
      | 1    | anno1 |
      | 2    | anno2 |
      | 3    | anno3 |

  Scenario:
    Given I use puts and step fails
      Announce with fail
       (RuntimeError)
      ./features/step_definitions/puts_steps.rb:18:in `/^I use puts and step
fails$/'
      features/f.feature:21:in `Given I use puts and step fails'
    And this step passes

  Scenario Outline:
    Given I use message <ann> in line <line> with result <result>

    Examples:
      | line | ann   | result |
      | 1    | anno1 | fail   |  Line: 1: anno1
       (RuntimeError)
      ./features/step_definitions/puts_steps.rb:13:in `/^I use message (.+)
in line (.+) (?:with result (.+))$/'
      features/f.feature:29:in `Given I use message anno1 in line 1 with
result fail'
      features/f.feature:25:in `Given I use message <ann> in line <line>
with result <result>'
      | 2    | anno2 | pass   |  Line: 2: anno2
```

Non-delayed messages feature (progress formatter)

When

I run `cucumber --format progress features/f.feature` 👍 (040ms)

Then

the output should contain: 👍 (000ms)

```
Ann
..
Multiple

Announce

Me
..UUU
Announce with fail
F-
Line: 1: anno1
F
Line: 2: anno2
.
```

Pretty output formatter

Background

Given

a file named "features/scenario_outline_with_undefined_steps.feature" with: 👍 (000ms)

```
Feature:

  Scenario Outline:
    Given this step is undefined

  Examples:
    |foo|
    |bar|
```

an scenario outline, one undefined step, one random example, expand flag on

When

I run `cucumber features/scenario_outline_with_undefined_steps.feature --format pretty --expand ` 👍 (017ms)

Then

it should pass 👍 (000ms)

when using a profile the output should include 'Using the default profile...'

And

a file named "cucumber.yml" with: 👍 (000ms)

```
  default: -r features
```

When

I run `cucumber --profile default --format pretty` 👍 (018ms)

Then

it should pass 👍 (000ms)

And

the output should contain: 👍 (000ms)

```
Using the default profile...
```

Hook output should be printed before hook exception

Given

the standard step definitions 👍 (000ms)

And

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
```

And

a file named "features/step_definitions/output_steps.rb" with: 👍 (000ms)

```
Before do
  puts "Before hook"
 end

AfterStep do
  puts "AfterStep hook"
end

After do
  puts "After hook"
raise "error"
end
```

When

I run `cucumber -q -f pretty features/test.feature` 👍 (015ms)

Then

the stderr should not contain anything 👍 (000ms)

Then

it should fail with: 👍 (000ms)

```
Feature:

  Scenario:
      Before hook
    Given this step passes
      AfterStep hook
      After hook
      error (RuntimeError)
      ./features/step_definitions/output_steps.rb:11:in `After'

Failing Scenarios:
cucumber features/test.feature:2

1 scenario (1 failed)
1 step (1 passed)
```

Profiles

In order to save time and prevent carpal tunnel syndrome
Cucumber users can save and reuse commonly used cucumber flags in a 'cucumber.yml' file.
These named arguments are called profiles and the yml file should be in the root of your project.
Any cucumber argument is valid in a profile. To see all the available flags type 'cucumber --help'
For more information about profiles please see the wiki:
http://wiki.github.com/cucumber/cucumber/cucumber.yml

Background

Basic App

Given

a file named "features/sample.feature" with: 👍 (000ms)

```
Feature: Sample
  Scenario: this is a test
    Given this step raises an error
```

And

an empty file named "features/support/env.rb" 👍 (000ms)

And

an empty file named "features/support/super_env.rb" 👍 (000ms)

And

the following profiles are defined: 👍 (000ms)

```
default: features/sample.feature --require features/support/env.rb -v
super: features/sample.feature --require features/support/super_env.rb -v
```

Explicitly defining a profile to run

When

I run `cucumber features/sample.feature --profile super` 👍 (014ms)

Then

the output should contain: 👍 (000ms)

```
Using the super profile...
```

And

exactly these files should be loaded: features/support/super_env.rb 👍 (000ms)

Explicitly defining a profile defined in an ERB formatted file

Given

the following profiles are defined: 👍 (000ms)

```
<% requires = "--require features/support/super_env.rb" %>
super: <%= "features/sample.feature #{requires} -v" %>
```

When

I run `cucumber features/sample.feature --profile super` 👍 (013ms)

Then

the output should contain: 👍 (000ms)

```
Using the super profile...
```

And

exactly these files should be loaded: features/support/super_env.rb 👍 (000ms)

Defining multiple profiles to run

When

I run `cucumber features/sample.feature --profile default --profile super` 👍 (013ms)

Then

the output should contain: 👍 (000ms)

```
Using the default and super profiles...
```

And

exactly these files should be loaded: features/support/env.rb, features/support/super_env.rb 👍 (000ms)

Arguments passed in but no profile specified

When

I run `cucumber -v` 👍 (009ms)

Then

the default profile should be used 👍 (000ms)

And

exactly these files should be loaded: features/support/env.rb 👍 (000ms)

Trying to use a missing profile

When

I run `cucumber -p foo` 👍 (004ms)

Then

the stderr should contain: 👍 (000ms)

```
Could not find profile: 'foo'

Defined profiles in cucumber.yml:
  * default
  * super
```

Disabling the default profile

When

I run `cucumber -v features/ -P` 👍 (006ms)

Then

the output should contain: 👍 (000ms)

```
Disabling profiles...
```

And

exactly these files should be loaded: features/support/env.rb, features/support/super_env.rb 👍 (000ms)

Disabling the default profile

When

I run `cucumber -v features/ --no-profile` 👍 (006ms)

Then

the output should contain: 👍 (000ms)

```
Disabling profiles...
```

And

exactly these files should be loaded: features/support/env.rb, features/support/super_env.rb 👍 (000ms)

Overriding the profile’s features to run

Given

a file named "features/another.feature" with: 👍 (000ms)

```
Feature: Just this one should be ran
```

When

I run `cucumber -p default features/another.feature` 👍 (007ms)

Then

exactly these features should be ran: features/another.feature 👍 (000ms)

Overriding the profile’s formatter

You will most likely want to define a formatter in your default formatter. However, you often want to run your features with a different formatter yet still use the other the other arguments in the profile. Cucumber will allow you to do this by giving precedence to the formatter specified on the command line and override the one in the profile.

Given

the following profiles are defined: 👍 (000ms)

```
default: features/sample.feature --require features/support/env.rb -v
--format profile
```

When

I run `cucumber features --format pretty` 👍 (008ms)

Then

the output should contain: 👍 (000ms)

```
Feature: Sample
```

Showing profiles when listing failing scenarios

Given

the standard step definitions 👍 (000ms)

When

I run `cucumber -q -p super -p default -f pretty features/sample.feature --require features/step_definitions/steps.rb` 👍 (015ms)

Then

it should fail with: 👍 (000ms)

```
cucumber -p super features/sample.feature:2
```

Showing profiles when listing failing scenarios

Given

the standard step definitions 👍 (000ms)

When

I run `cucumber -q -p super -p default -f progress features/sample.feature --require features/step_definitions/steps.rb` 👍 (016ms)

Then

it should fail with: 👍 (000ms)

```
cucumber -p super features/sample.feature:2
```

Progress output formatter

Background

Given

a file named "features/scenario_outline_with_undefined_steps.feature" with: 👍 (000ms)

```
Feature:

  Scenario Outline:
    Given this step is undefined

  Examples:
    |foo|
    |bar|
```

an scenario outline, one undefined step, one random example, expand flag on

When

I run `cucumber features/scenario_outline_with_undefined_steps.feature --format progress --expand ` 👍 (009ms)

Then

it should pass 👍 (000ms)

when using a profile the output should include 'Using the default profile...'

And

a file named "cucumber.yml" with: 👍 (000ms)

```
  default: -r features
```

When

I run `cucumber --profile default --format progress` 👍 (012ms)

Then

it should pass 👍 (000ms)

And

the output should contain: 👍 (000ms)

```
Using the default profile...
```

Rake task

In order to ease the development process
As a developer and CI server administrator
Cucumber features should be executable via Rake

Background

tags: @spawn

And

a file named "features/missing_step_definitions.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario: Wanted
    Given I want to run this

  Scenario: Unwanted
    Given I don't want this ran
```

rake task with a defined profile

tags: @spawn,@spawn

Given

the following profile is defined: 👍 (000ms)

```
foo: --quiet --no-color features/missing_step_definitions.feature:3
```

And

a file named "Rakefile" with: 👍 (000ms)

```
require 'cucumber/rake/task'

Cucumber::Rake::Task.new do |t|
  t.profile = "foo"
end
```

When

I run `rake cucumber` 👍 (01s 206ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Sample

  Scenario: Wanted
    Given I want to run this

1 scenario (1 undefined)
1 step (1 undefined)
```

rake task without a profile

tags: @spawn,@spawn

Given

a file named "Rakefile" with: 👍 (000ms)

```
require 'cucumber/rake/task'

Cucumber::Rake::Task.new do |t|
  t.cucumber_opts = %w{--quiet --no-color}
end
```

When

I run `rake cucumber` 👍 (01s 207ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Sample

  Scenario: Wanted
    Given I want to run this

  Scenario: Unwanted
    Given I don't want this ran

2 scenarios (2 undefined)
2 steps (2 undefined)
```

rake task with a defined profile and cucumber_opts

tags: @spawn,@spawn

Given

the following profile is defined: 👍 (000ms)

```
bar: ['features/missing_step_definitions.feature:3']
```

And

a file named "Rakefile" with: 👍 (000ms)

```
require 'cucumber/rake/task'

Cucumber::Rake::Task.new do |t|
  t.profile = "bar"
  t.cucumber_opts = %w{--quiet --no-color}
end
```

When

I run `rake cucumber` 👍 (01s 207ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Sample

  Scenario: Wanted
    Given I want to run this

1 scenario (1 undefined)
1 step (1 undefined)
```

respect requires

tags: @spawn,@spawn

Given

an empty file named "features/support/env.rb" 👍 (000ms)

And

an empty file named "features/support/dont_require_me.rb" 👍 (000ms)

And

the following profile is defined: 👍 (000ms)

```
no_bomb: features/missing_step_definitions.feature:3 --require
features/support/env.rb --verbose
```

And

a file named "Rakefile" with: 👍 (000ms)

```
require 'cucumber/rake/task'

Cucumber::Rake::Task.new do |t|
  t.profile = "no_bomb"
  t.cucumber_opts = %w{--quiet --no-color}
end
```

When

I run `rake cucumber` 👍 (01s 106ms)

Then

it should pass 👍 (000ms)

And

the output should not contain: 👍 (000ms)

```
  * features/support/dont_require_me.rb
```

feature files with spaces

tags: @spawn,@spawn

Given

a file named "features/spaces are nasty.feature" with: 👍 (000ms)

```
Feature: The futures green

  Scenario: Orange
    Given this is missing
```

And

a file named "Rakefile" with: 👍 (000ms)

```
require 'cucumber/rake/task'

Cucumber::Rake::Task.new do |t|
  t.cucumber_opts = %w{--quiet --no-color}
end
```

When

I run `rake cucumber` 👍 (01s 108ms)

Then

it should pass with: 👍 (001ms)

```
Feature: The futures green

  Scenario: Orange
    Given this is missing
```

Raketask

In order to use cucumber’s rake task
As a Cuker
I do not want to see rake’s backtraces when it fails
Also I want to get zero exit status code on failures
And non-zero exit status code when it pases

Background

tags: @spawn

Given

the standard step definitions 👍 (000ms)

Given

a file named "features/passing_and_failing.feature" with: 👍 (000ms)

```
Feature: Sample

  Scenario: Passing
    Given this step passes

  Scenario: Failing
    Given this step raises an error
```

Given

a file named "Rakefile" with: 👍 (000ms)

```
  require 'cucumber/rake/task'

  SAMPLE_FEATURE_FILE = 'features/passing_and_failing.feature'

  Cucumber::Rake::Task.new(:pass) do |t|
    t.cucumber_opts = "#{SAMPLE_FEATURE_FILE}:3"
  end

  Cucumber::Rake::Task.new(:fail) do |t|
    t.cucumber_opts = "#{SAMPLE_FEATURE_FILE}:6"
  end
```

Passing feature

tags: @spawn,@spawn

When

I run `bundle exec rake pass` 👍 (01s 608ms)

Then

the exit status should be 0 👍 (000ms)

Failing feature

tags: @spawn,@spawn

When

I run `bundle exec rake fail` 👍 (02s 210ms)

Then

the exit status should be 1 👍 (000ms)

But

the output should not contain "rake aborted!" 👍 (000ms)

Randomize

Use the `--order random` switch to run scenarios in random order.

This is especially helpful for detecting situations where you have state
leaking between scenarios, which can cause flickering or fragile tests.

If you do find a randmon run that exposes dependencies between your tests,
you can reproduce that run by using the seed that's printed at the end of
the test run.

Background

Given

a file named "features/bad_practice.feature" with: 👍 (000ms)

```
Feature: Bad practice

  Scenario: Set state
    Given I set some state

  Scenario: Depend on state
    When I depend on the state
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given(/^I set some state$/) do
  $global_state = "set"
end

Given(/^I depend on the state$/) do
  raise "I expect the state to be set!" unless $global_state == "set"
end
```

Run scenarios in order

When

I run `cucumber` 👍 (006ms)

Then

it should pass 👍 (000ms)

Run scenarios randomized

tags: @spawn

When

I run `cucumber --order random:41515` 👍 (705ms)

Then

it should fail 👍 (000ms)

And

the stdout should contain: 👍 (000ms)

```
Randomized with seed 41515
```

Requiring extra step files

Cucumber allows you to require extra files using the `-r` option.

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: Sample
  Scenario: Sample
    Given found in extra file
```

And

a file named "tmp/extras.rb" with: 👍 (000ms)

```
Given(/^found in extra file$/) { }
```

When

I run `cucumber -q -r tmp/extras.rb features/test.feature` 👍 (011ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Sample

  Scenario: Sample
    Given found in extra file

1 scenario (1 passed)
1 step (1 passed)
```

Rerun formatter

The rerun formatter writes an output that's perfect for passing to Cucumber when you want to rerun only the scenarios that prevented the exit code to be zero.

You can save off the rerun output to a file by using it like this:

```
cucumber -f rerun --out .cucumber.rerun
```

Now you can pass that file's content to Cucumber to tell it which scenarios to run:

```
cucumber `cat .cucumber.rerun`
```

This is useful when debugging in a large suite of features.

Background

Given

the standard step definitions 👍 (000ms)

Exit code is zero

Given

a file named "features/mixed.feature" with: 👍 (000ms)

```
Feature: Mixed

  Scenario:
    Given this step is undefined

  Scenario:
    Given this step is pending

  Scenario:
    Given this step passes
```

When

I run `cucumber -f rerun` 👍 (013ms)

Then

it should pass with exactly: 👍 (000ms)

Exit code is zero in the dry-run mode

Given

a file named "features/mixed.feature" with: 👍 (000ms)

```
Feature: Mixed

  Scenario:
    Given this step fails

  Scenario:
    Given this step is undefined

  Scenario:
    Given this step is pending

  Scenario:
    Given this step passes
```

And

a file named "features/all_good.feature" with: 👍 (000ms)

```
Feature: All good

  Scenario:
    Given this step passes
```

When

I run `cucumber -f rerun --dry-run` 👍 (016ms)

Then

it should pass with exactly: 👍 (000ms)

Exit code is not zero, regular scenario

Given

a file named "features/mixed.feature" with: 👍 (000ms)

```
Feature: Mixed

  Scenario:
    Given this step fails

  Scenario:
    Given this step is undefined

  Scenario:
    Given this step is pending

  Scenario:
    Given this step passes
```

And

a file named "features/all_good.feature" with: 👍 (000ms)

```
Feature: All good

  Scenario:
    Given this step passes
```

When

I run `cucumber -f rerun --strict` 👍 (016ms)

Then

it should fail with exactly: 👍 (000ms)

```
features/mixed.feature:3:6:9
```

Exit code is not zero, scenario outlines

For details see https://github.com/cucumber/cucumber/issues/57

Given

a file named "features/one_passing_one_failing.feature" with: 👍 (000ms)

```
Feature: One passing example, one failing example

  Scenario Outline:
    Given this step <status>

  Examples:
    | status |
    | passes |
    | fails  |
```

When

I run `cucumber -f rerun` 👍 (010ms)

Then

it should fail with: 👍 (000ms)

```
features/one_passing_one_failing.feature:9
```

Exit code is not zero, failing background

Given

a file named "features/failing_background.feature" with: 👍 (000ms)

```
Feature: Failing background sample

  Background:
    Given this step fails

  Scenario: failing background
    Then this step passes

  Scenario: another failing background
    Then this step passes
```

When

I run `cucumber -f rerun` 👍 (012ms)

Then

it should fail with: 👍 (000ms)

```
features/failing_background.feature:6:9
```

Exit code is not zero, failing background with scenario outline

Given

a file named "features/failing_background_outline.feature" with: 👍 (000ms)

```
Feature: Failing background sample with scenario outline

  Background:
    Given this step fails

  Scenario Outline:
    Then this step <status>

  Examples:
    | status |
    | passes |
    | passes |
```

When

I run `cucumber features/failing_background_outline.feature -r features -f rerun` 👍 (011ms)

Then

it should fail with: 👍 (000ms)

```
features/failing_background_outline.feature:11:12
```

Exit code is not zero, scenario outlines with expand

For details see https://github.com/cucumber/cucumber/issues/503

Given

a file named "features/one_passing_one_failing.feature" with: 👍 (000ms)

```
  Feature: One passing example, one failing example

    Scenario Outline:
      Given this step <status>

    Examples:
      | status |
      | passes |
      | fails  |
```

When

I run `cucumber --expand -f rerun` 👍 (012ms)

Then

it should fail with: 👍 (000ms)

```
features/one_passing_one_failing.feature:9
```

Run Cli::Main with existing Runtime

This is the API that Spork uses. It creates an existing runtime then
calls load_programming_language('rb') on it to load the RbDsl.
When the process forks, Spork them passes the runtime to Cli::Main to
run it.

Run a single feature

tags: @spawn,@spawn

Given

the standard step definitions 👍 (000ms)

Given

a file named "features/success.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes
```

When

I run the following Ruby code: 👍 (607ms)

```
require 'cucumber'
runtime = Cucumber::Runtime.new
runtime.load_programming_language('rb')
Cucumber::Cli::Main.new([]).execute!(runtime)
```

Then

it should pass 👍 (000ms)

And

the output should contain: 👍 (005ms)

```
Given this step passes
```

[[Run-feature-elements-matching-a-name-with---name/-n, Run feature elements matching a name with --name/-n]]
=== **Run feature elements matching a name with --name/-n**

The `--name NAME` option runs only scenarios which match a certain
name. The NAME can be a substring of the names of Features, Scenarios,
Scenario Outlines or Example blocks.

Background

Given

a file named "features/first.feature" with: 👍 (000ms)

```
Feature: first feature
  Scenario: foo first
    Given missing
  Scenario: bar first
    Given missing
```

Given

a file named "features/second.feature" with: 👍 (000ms)

```
Feature: second
  Scenario: foo second
    Given missing
  Scenario: bar second
    Given missing
```

Given

a file named "features/outline.feature" with: 👍 (000ms)

```
Feature: outline
  Scenario Outline: baz outline
    Given outline step <name>

    Examples: quux example
      | name |
      | a    |
      | b    |
```

Matching Feature names

When

I run `cucumber -q --name feature` 👍 (012ms)

Then

it should pass with: 👍 (000ms)

```
Feature: first feature

  Scenario: foo first
    Given missing

  Scenario: bar first
    Given missing

2 scenarios (2 undefined)
2 steps (2 undefined)
```

Matching Scenario names

When

I run `cucumber -q --name foo` 👍 (011ms)

Then

it should pass with: 👍 (000ms)

```
Feature: first feature

  Scenario: foo first
    Given missing

Feature: second

  Scenario: foo second
    Given missing

2 scenarios (2 undefined)
2 steps (2 undefined)
```

Matching Scenario Outline names

When

I run `cucumber -q --name baz` 👍 (012ms)

Then

it should pass with: 👍 (000ms)

```
Feature: outline

  Scenario Outline: baz outline
    Given outline step <name>

    Examples: quux example
      | name |
      | a    |
      | b    |

2 scenarios (2 undefined)
2 steps (2 undefined)
```

Matching Example block names

When

I run `cucumber -q --name quux` 👍 (010ms)

Then

it should pass with: 👍 (000ms)

```
Feature: outline

  Scenario Outline: baz outline
    Given outline step <name>

    Examples: quux example
      | name |
      | a    |
      | b    |

2 scenarios (2 undefined)
2 steps (2 undefined)
```

Run specific scenarios

You can choose to run a specific scenario using the file:line format,
or you can pass in a file with a list of scenarios using @-notation.

The line number can fall anywhere within the body of a scenario, including
steps, tags, comments, description, data tables or doc strings.

For scenario outlines, if the line hits one example row, just that one
will be run. Otherwise all examples in the table or outline will be run.

Background

Given

the standard step definitions 👍 (000ms)

Two scenarios, run just one of them

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:

  Scenario: Miss
    Given this step is undefined

  Scenario: Hit
    Given this step passes
```

When

I run `cucumber features/test.feature:7 --format pretty --quiet` 👍 (008ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature:

  Scenario: Hit
    Given this step passes

1 scenario (1 passed)
1 step (1 passed)
```

Use @-notation to specify a file containing feature file list

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: Sample
  Scenario: Passing
    Given this step passes
```

And

a file named "list-of-features.txt" with: 👍 (000ms)

```
features/test.feature:2
```

When

I run `cucumber -q @list-of-features.txt` 👍 (006ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Sample

  Scenario: Passing
    Given this step passes

1 scenario (1 passed)
1 step (1 passed)
```

Specify order of scenarios

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given this step passes

  Scenario:
    Given this step fails
```

When

I run `cucumber features/test.feature:5 features/test.feature:3 -f progress` 👍 (009ms)

Then

it should fail with: 👍 (000ms)

```
F.
```

Running multiple formatters

When running cucumber, you are able to using multiple different
formatters and redirect the output to text files.
Two formatters cannot both print to the same file (or to STDOUT)

Background

tags: @spawn

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: Lots of undefined

  Scenario: Implement me
    Given it snows in Sahara
    Given it's 40 degrees in Norway
    And it's 40 degrees in Norway
    When I stop procrastinating
    And there is world peace
```

Multiple formatters and outputs

tags: @spawn,@spawn

When

I run `cucumber --no-color --format progress --out progress.txt --format pretty --out pretty.txt --no-source --dry-run --no-snippets features/test.feature` 👍 (606ms)

Then

the stderr should not contain anything 👍 (006ms)

Then

the file "progress.txt" should contain: 👍 (000ms)

```
UUUUU

1 scenario (1 undefined)
5 steps (5 undefined)
```

And

the file "pretty.txt" should contain: 👍 (000ms)

```
Feature: Lots of undefined

  Scenario: Implement me
    Given it snows in Sahara
    Given it's 40 degrees in Norway
    And it's 40 degrees in Norway
    When I stop procrastinating
    And there is world peace

1 scenario (1 undefined)
5 steps (5 undefined)
```

Two formatters to stdout

tags: @spawn,@spawn

When

I run `cucumber -f progress -f pretty features/test.feature` 👍 (607ms)

Then

it should fail with: 👍 (000ms)

```
All but one formatter must use --out, only one can print to each stream (or
STDOUT) (RuntimeError)
```

Two formatters to stdout when using a profile

tags: @spawn,@spawn

Given

the following profiles are defined: 👍 (000ms)

```
default: -q
```

When

I run `cucumber -f progress -f pretty features/test.feature` 👍 (606ms)

Then

it should fail with: 👍 (000ms)

```
All but one formatter must use --out, only one can print to each stream (or
STDOUT) (RuntimeError)
```

Scenario outlines

Copying and pasting scenarios to use different values quickly becomes tedious and repetitive. Scenario outlines allow us to more concisely express these examples through the use of a template with placeholders, using Scenario Outline, Examples with tables and < > delimited parameters.

The Scenario Outline steps provide a template which is never directly run. A Scenario Outline is run once for each row in the Examples section beneath it (not counting the first row).

The way this works is via placeholders. Placeholders must be contained within < > in the Scenario Outline's steps - see the examples below.

IMPORTANT: Your step definitions will never have to match a placeholder. They will need to match the values that will replace the placeholder.

Background

tags: @spawn

Given

a file named "features/outline_sample.feature" with: 👍 (000ms)

```
Feature: Outline Sample

  Scenario: I have no steps

  Scenario Outline: Test state
    Given <state> without a table
    Given <other_state> without a table
  Examples: Rainbow colours
    | state    | other_state |
    | missing  | passing     |
    | passing  | passing     |
    | failing  | passing     |
Examples:Only passing
    | state    | other_state |
    | passing  | passing     |
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given(/^passing without a table$/) { }
Given(/^failing without a table$/) { raise RuntimeError }
```

Run scenario outline with filtering on outline name

tags: @spawn,@spawn

When

I run `cucumber -q features/outline_sample.feature` 👍 (706ms)

Then

it should fail with: 👍 (001ms)

```
Feature: Outline Sample

  Scenario: I have no steps

  Scenario Outline: Test state
    Given <state> without a table
    Given <other_state> without a table

    Examples: Rainbow colours
      | state   | other_state |
      | missing | passing     |
      | passing | passing     |
      | failing | passing     |
      RuntimeError (RuntimeError)
      ./features/step_definitions/steps.rb:2:in `/^failing without a
table$/'
      features/outline_sample.feature:12:in `Given failing without a table'
      features/outline_sample.feature:6:in `Given <state> without a table'

    Examples: Only passing
      | state   | other_state |
      | passing | passing     |

Failing Scenarios:
cucumber features/outline_sample.feature:12

5 scenarios (1 failed, 1 undefined, 3 passed)
8 steps (1 failed, 2 skipped, 1 undefined, 4 passed)
```

Run scenario outline steps only

tags: @spawn,@spawn

When

I run `cucumber -q features/outline_sample.feature:7` 👍 (607ms)

Then

it should fail with: 👍 (000ms)

```
Feature: Outline Sample

  Scenario Outline: Test state
    Given <state> without a table
    Given <other_state> without a table

    Examples: Rainbow colours
      | state   | other_state |
      | missing | passing     |
      | passing | passing     |
      | failing | passing     |
      RuntimeError (RuntimeError)
      ./features/step_definitions/steps.rb:2:in `/^failing without a
table$/'
      features/outline_sample.feature:12:in `Given failing without a table'
      features/outline_sample.feature:6:in `Given <state> without a table'

    Examples: Only passing
      | state   | other_state |
      | passing | passing     |

Failing Scenarios:
cucumber features/outline_sample.feature:12

4 scenarios (1 failed, 1 undefined, 2 passed)
8 steps (1 failed, 2 skipped, 1 undefined, 4 passed)
```

Run single failing scenario outline table row

tags: @spawn,@spawn

When

I run `cucumber -q features/outline_sample.feature:12` 👍 (504ms)

Then

it should fail with: 👍 (001ms)

```
Feature: Outline Sample

  Scenario Outline: Test state
    Given <state> without a table
    Given <other_state> without a table

    Examples: Rainbow colours
      | state   | other_state |
      | failing | passing     |
      RuntimeError (RuntimeError)
      ./features/step_definitions/steps.rb:2:in `/^failing without a
table$/'
      features/outline_sample.feature:12:in `Given failing without a table'
      features/outline_sample.feature:6:in `Given <state> without a table'

Failing Scenarios:
cucumber features/outline_sample.feature:12

1 scenario (1 failed)
2 steps (1 failed, 1 skipped)
```

Run all with progress formatter

tags: @spawn,@spawn

When

I run `cucumber -q --format progress features/outline_sample.feature` 👍 (606ms)

Then

it should fail with exactly: 👍 (000ms)

```
U-..F-..

(::) failed steps (::)

RuntimeError (RuntimeError)
./features/step_definitions/steps.rb:2:in `/^failing without a table$/'
features/outline_sample.feature:12:in `Given failing without a table'
features/outline_sample.feature:6:in `Given <state> without a table'

Failing Scenarios:
cucumber features/outline_sample.feature:12

5 scenarios (1 failed, 1 undefined, 3 passed)
8 steps (1 failed, 2 skipped, 1 undefined, 4 passed)
```

Scenario outlines --expand option

In order to make it easier to write certain editor plugins and also
for some people to understand scenarios, Cucumber will expand examples
in outlines if you add the `--expand` option when running them.

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:
  Scenario Outline:
    Given the secret code is <code>
    When I guess <guess>
    Then I am <verdict>

  Examples:
    | code | guess | verdict |
    | blue | blue  | right   |
    | red  | blue  | wrong   |
```

When

I run `cucumber -i -q --expand` 👍 (013ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should pass with: 👍 (000ms)

```
Feature:

  Scenario Outline:
    Given the secret code is <code>
    When I guess <guess>
    Then I am <verdict>

    Examples:

      Scenario: | blue | blue | right |
        Given the secret code is blue
        When I guess blue
        Then I am right

      Scenario: | red | blue | wrong |
        Given the secret code is red
        When I guess blue
        Then I am wrong

2 scenarios (2 undefined)
6 steps (6 undefined)
```

Set up a default load path

When you're developing a gem, it's convenient if your project's `lib` directory is already in the load path. Cucumber does this for you.

./lib is included in the $LOAD_PATH

Given

a file named "features/support/env.rb" with: 👍 (000ms)

```
require 'something'
```

And

a file named "lib/something.rb" with: 👍 (000ms)

```
class Something
end
```

When

I run `cucumber` 👍 (009ms)

Then

it should pass 👍 (000ms)

Showing differences to expected output

Cucumber will helpfully show you the expectation error that your
testing library gives you, in the context of the failing scenario.
When using RSpec, for example, this will show the difference between
the expected and the actual output.

Run single failing scenario with default diff enabled

Given

a file named "features/failing_expectation.feature" with: 👍 (000ms)

```
Feature: Failing expectation

  Scenario: Failing expectation
    Given failing expectation
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given /^failing expectation$/ do x=1
  expect('this').to eq 'that'
end
```

When

I run `cucumber -q features/failing_expectation.feature` 👍 (022ms)

Then

it should fail with: 👍 (000ms)

```
Feature: Failing expectation

  Scenario: Failing expectation
    Given failing expectation

      expected: "that"
           got: "this"

      (compared using ==)
       (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/steps.rb:2:in `/^failing expectation$/'
      features/failing_expectation.feature:4:in `Given failing expectation'

Failing Scenarios:
cucumber features/failing_expectation.feature:3

1 scenario (1 failed)
1 step (1 failed)
```

Skip Scenario

With a passing step

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: test
  Scenario: test
    Given this step says to skip
    And this step passes
```

And

the standard step definitions 👍 (000ms)

And

a file named "features/step_definitions/skippy.rb" with: 👍 (000ms)

```
Given /skip/ do
  skip_this_scenario
end
```

When

I run `cucumber -q` 👍 (011ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: test

  Scenario: test
    Given this step says to skip
    And this step passes

1 scenario (1 skipped)
2 steps (2 skipped)
```

Use legacy API from a hook

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: test
  Scenario: test
    Given this step passes
    And this step passes
```

And

the standard step definitions 👍 (000ms)

And

a file named "features/support/hook.rb" with: 👍 (000ms)

```
Before do |scenario|
  scenario.skip_invoke!
end
```

When

I run `cucumber -q` 👍 (011ms)

Then

it should pass with: 👍 (000ms)

```
Feature: test

  Scenario: test
    Given this step passes
    And this step passes

1 scenario (1 skipped)
2 steps (2 skipped)
```

Snippets

Cucumber helpfully prints out any undefined step definitions as a code
snippet suggestion, which you can then paste into a step definitions
file of your choosing.

Snippet for undefined step with a pystring

Given

a file named "features/undefined_steps.feature" with: 👍 (000ms)

```
Feature:
Scenario: pystring
  Given a pystring
  """
    example with <html> entities
  """
  When a simple when step
  And another when step
  Then a simple then step
```

When

I run `cucumber features/undefined_steps.feature -s` 👍 (009ms)

Then

the output should contain: 👍 (000ms)

```
Given(/^a pystring$/) do |string|
  pending # Write code here that turns the phrase above into concrete
actions
end

When(/^a simple when step$/) do
  pending # Write code here that turns the phrase above into concrete
actions
end

When(/^another when step$/) do
  pending # Write code here that turns the phrase above into concrete
actions
end

Then(/^a simple then step$/) do
  pending # Write code here that turns the phrase above into concrete
actions
end
```

Snippet for undefined step with a step table

Given

a file named "features/undefined_steps.feature" with: 👍 (000ms)

```
Feature:
Scenario: table
  Given a table
    | table |
    |example|
```

When

I run `cucumber features/undefined_steps.feature -s` 👍 (007ms)

Then

the output should contain: 👍 (000ms)

```
Given(/^a table$/) do |table|
  # table is a Cucumber::Core::Ast::DataTable
  pending # Write code here that turns the phrase above into concrete
actions
end
```

Snippets message

If a step doesn't match, Cucumber will ask the wire server to return a snippet of code for a step definition.

Background

tags: @wire

Given

a file named "features/wired.feature" with: 👍 (000ms)

```
Feature: High strung
  Scenario: Wired
    Given we're all wired
```

And

a file named "features/step_definitions/some_remote_place.wire" with: 👍 (000ms)

```
host: localhost
port: 54321
```

Wire server returns snippets for a step that didn’t match

tags: @wire,@wire,@spawn

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f pretty` 👍 (908ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should pass with: 👍 (001ms)

```
Feature: High strung

  Scenario: Wired         # features/wired.feature:2
    Given we're all wired # features/wired.feature:3

1 scenario (1 undefined)
1 step (1 undefined)
```

And

the output should contain: 👍 (000ms)

```
You can implement step definitions for undefined steps with these snippets:

foo()
  bar;
baz
```

State

You can pass state between step by setting instance variables,
but those instance variables will be gone when the next scenario runs.

Set an ivar in one scenario, use it in the next step

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given I have set @flag = true
    Then @flag should be true

  Scenario:
    Then @flag should be nil
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given /set @flag/ do
  @flag = true
end
Then /flag should be true/ do
  expect(@flag).to be_truthy
end
Then /flag should be nil/ do
  expect(@flag).to be_nil
end
```

When

I run `cucumber` 👍 (014ms)

Then

it should pass 👍 (000ms)

Step matches message

When the features have been parsed, Cucumber will send a `step_matches` message to ask the wire server if it can match a step name. This happens for each of the steps in each of the features.

The wire server replies with an array of StepMatch objects.

When each StepMatch is returned, it contains the following data:

- `id` - identifier for the step definition to be used later when if it needs to be invoked. The identifier can be any string value and is simply used for the wire server's own reference.
- `args` - any argument values as captured by the wire end's own regular expression (or other argument matching) process.

Background

tags: @wire

Given

a file named "features/wired.feature" with: 👍 (000ms)

```
Feature: High strung
  Scenario: Wired
    Given we're all wired
```

And

a file named "features/step_definitions/some_remote_place.wire" with: 👍 (000ms)

```
host: localhost
port: 54321
```

Dry run finds no step match

tags: @wire,@wire

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber --dry-run --no-snippets -f progress` 👍 (061ms)

And

it should pass with: 👍 (000ms)

```
U

1 scenario (1 undefined)
1 step (1 undefined)
```

Dry run finds a step match

tags: @wire,@wire

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber --dry-run -f progress` 👍 (024ms)

And

it should pass with: 👍 (000ms)

```
-

1 scenario (1 skipped)
1 step (1 skipped)
```

Step matches returns details about the remote step definition

tags: @wire,@wire

Optionally, the StepMatch can also contain a source reference, and a native
regexp string which will be used by some formatters.

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (001ms)

When

I run `cucumber -f stepdefs --dry-run` 👍 (015ms)

Then

it should pass with: 👍 (000ms)

```
-

we.*    # MyApp.MyClass:123

1 scenario (1 skipped)
1 step (1 skipped)
```

And

the stderr should not contain anything 👍 (000ms)

Strict mode

Using the `--strict` flag will cause cucumber to fail unless all the
step definitions have been defined.

Background

Given

a file named "features/missing.feature" with: 👍 (000ms)

```
Feature: Missing
  Scenario: Missing
    Given this step passes
```

And

a file named "features/pending.feature" with: 👍 (000ms)

```
Feature: Pending
  Scenario: Pending
    Given this step is pending
```

Fail with --strict due to undefined step

When

I run `cucumber -q features/missing.feature --strict` 👍 (016ms)

Then

it should fail with: 👍 (000ms)

```
Feature: Missing

  Scenario: Missing
    Given this step passes
      Undefined step: "this step passes" (Cucumber::Undefined)
      features/missing.feature:3:in `Given this step passes'

1 scenario (1 undefined)
1 step (1 undefined)
```

Fail with --strict due to pending step

Given

the standard step definitions 👍 (000ms)

When

I run `cucumber -q features/pending.feature --strict` 👍 (015ms)

Then

it should fail with: 👍 (000ms)

```
Feature: Pending

  Scenario: Pending
    Given this step is pending
      TODO (Cucumber::Pending)
      ./features/step_definitions/steps.rb:3:in `/^this step is pending$/'
      features/pending.feature:3:in `Given this step is pending'

1 scenario (1 pending)
1 step (1 pending)
```

Succeed with --strict

Given

the standard step definitions 👍 (000ms)

When

I run `cucumber -q features/missing.feature --strict` 👍 (010ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Missing

  Scenario: Missing
    Given this step passes

1 scenario (1 passed)
1 step (1 passed)
```

Table diffing

To allow you to more easily compare data in tables, you are able to easily diff a table with expected data and see the diff in your output.

Extra row

Given

a file named "features/tables.feature" with: 👍 (000ms)

```
Feature: Tables
  Scenario: Extra row
    Then the table should be:
      | x | y |
      | a | b |
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Then /the table should be:/ do |expected| x=1
  expected.diff!(table(%{
    | x | y |
    | a | c |
  }))
end
```

When

I run `cucumber features/tables.feature` 👍 (015ms)

Then

it should fail with exactly: 👍 (000ms)

```
Feature: Tables

  Scenario: Extra row         # features/tables.feature:2
    Then the table should be: # features/step_definitions/steps.rb:1
      | x | y |
      | a | b |
      Tables were not identical:

        |     x |     y |
        | (-) a | (-) b |
        | (+) a | (+) c |
       (Cucumber::MultilineArgument::DataTable::Different)
      ./features/step_definitions/steps.rb:2:in `/the table should be:/'
      features/tables.feature:3:in `Then the table should be:'

Failing Scenarios:
cucumber features/tables.feature:2 # Scenario: Extra row

1 scenario (1 failed)
1 step (1 failed)
0m0.012s
```

Tag logic

In order to conveniently run subsets of features
As a Cuker
I want to select features using logical AND/OR of tags

Background

Given

a file named "features/test.feature" with: 👍 (000ms)

```
@feature
Feature: Sample

  @one @three
  Scenario: Example
    Given passing

  @one
  Scenario: Another Example
    Given passing

  @three
  Scenario: Yet another Example
    Given passing

  @ignore
  Scenario: And yet another Example
```

ANDing tags

When

I run `cucumber -q -t @one -t @three features/test.feature` 👍 (008ms)

Then

it should pass with: 👍 (000ms)

```
@feature
Feature: Sample

  @one @three
  Scenario: Example
    Given passing

1 scenario (1 undefined)
1 step (1 undefined)
```

ORing tags

When

I run `cucumber -q -t @one,@three features/test.feature` 👍 (010ms)

Then

it should pass with: 👍 (000ms)

```
@feature
Feature: Sample

  @one @three
  Scenario: Example
    Given passing

  @one
  Scenario: Another Example
    Given passing

  @three
  Scenario: Yet another Example
    Given passing

3 scenarios (3 undefined)
3 steps (3 undefined)
```

Negative tags

When

I run `cucumber -q -t ~@three features/test.feature` 👍 (008ms)

Then

it should pass with: 👍 (000ms)

```
@feature
Feature: Sample

  @one
  Scenario: Another Example
    Given passing

  @ignore
  Scenario: And yet another Example

2 scenarios (1 undefined, 1 passed)
1 step (1 undefined)
```

Run with limited tag count, blowing it on scenario

When

I run `cucumber -q --no-source --tags @one:1 features/test.feature` 👍 (005ms)

Then

it fails before running features with: 👍 (000ms)

```
@one occurred 2 times, but the limit was set to 1
  features/test.feature:5
  features/test.feature:9
```

Run with limited tag count, blowing it via feature inheritance

When

I run `cucumber -q --no-source --tags @feature:1 features/test.feature` 👍 (005ms)

Then

it fails before running features with: 👍 (000ms)

```
@feature occurred 4 times, but the limit was set to 1
  features/test.feature:5
  features/test.feature:9
  features/test.feature:13
  features/test.feature:17
```

Run with limited tag count using negative tag, blowing it via a tag that is not run

When

I run `cucumber -q --no-source --tags ~@one:1 features/test.feature` 👍 (004ms)

Then

it fails before running features with: 👍 (000ms)

```
@one occurred 2 times, but the limit was set to 1
```

Limiting with tags which do not exist in the features

Originally added to check [Lighthouse bug #464](https://rspec.lighthouseapp.com/projects/16211/tickets/464).

When

I run `cucumber -q -t @i_dont_exist features/test.feature` 👍 (004ms)

Then

it should pass 👍 (000ms)

Tagged hooks

Background

Given

the standard step definitions 👍 (000ms)

And

a file named "features/support/hooks.rb" with: 👍 (000ms)

```
Before('~@no-boom') do
  raise 'boom'
end
```

And

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: With and without hooks
  Scenario: using hook
    Given this step passes

  @no-boom
  Scenario: omitting hook
    Given this step passes

  Scenario Outline: omitting hook on specified examples
    Given this step passes

    Examples:
    | Value       |
    | Irrelevant  |

    @no-boom
    Examples:
    | Value           |
    | Also Irrelevant |
```

omit tagged hook

When

I run `cucumber features/f.feature:2` 👍 (016ms)

Then

it should fail with exactly: 👍 (000ms)

```
Feature: With and without hooks

  Scenario: using hook     # features/f.feature:2
  boom (RuntimeError)
  ./features/support/hooks.rb:2:in `Before'
    Given this step passes # features/step_definitions/steps.rb:1

Failing Scenarios:
cucumber features/f.feature:2 # Scenario: using hook

1 scenario (1 failed)
1 step (1 skipped)
0m0.012s
```

omit tagged hook

When

I run `cucumber features/f.feature:6` 👍 (010ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: With and without hooks

  @no-boom
  Scenario: omitting hook  # features/f.feature:6
    Given this step passes # features/step_definitions/steps.rb:1

1 scenario (1 passed)
1 step (1 passed)
0m0.012s
```

Omit example hook

When

I run `cucumber features/f.feature:12` 👍 (013ms)

Then

it should fail with exactly: 👍 (000ms)

```
Feature: With and without hooks

  Scenario Outline: omitting hook on specified examples #
features/f.feature:9
    Given this step passes                               #
features/f.feature:10

    Examples:
      | Value      |
      boom (RuntimeError)
      ./features/support/hooks.rb:2:in `Before'
      | Irrelevant |

Failing Scenarios:
cucumber features/f.feature:14 # Scenario Outline: omitting hook on
specified examples, Examples (#1)

1 scenario (1 failed)
1 step (1 skipped)
0m0.012s
```

Transforms

If you see certain phrases repeated over and over in your step definitions, you can use transforms to factor out that duplication, and make your step definitions simpler.

Background

Let’s just create a simple feature for testing out Transforms.
We also have a Person class that we need to be able to build.

Given

a file named "features/foo.feature" with: 👍 (000ms)

```
Feature:
  Scenario:
    Given a Person aged 15 with blonde hair
```

And

a file named "features/support/person.rb" with: 👍 (000ms)

```
class Person
  attr_accessor :age

  def to_s
    "I am #{age} years old"
  end
end
```

Basic Transform

This is the most basic way to use a transform. Notice that the regular expression is pretty much duplicated.

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Transform(/a Person aged (\d+)/) do |age|
  person = Person.new
  person.age = age.to_i
  person
end

Given /^(a Person aged \d+) with blonde hair$/ do |person|
  expect(person.age).to eq 15
end
```

When

I run `cucumber features/foo.feature` 👍 (009ms)

Then

it should pass 👍 (000ms)

Re-use Transform's Regular Expression

If you keep a reference to the transform, you can use it in your regular expressions to avoid repeating the regular expression.

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
A_PERSON = Transform(/a Person aged (\d+)/) do |age|
  person = Person.new
  person.age = age.to_i
  person
end

Given /^(#{A_PERSON}) with blonde hair$/ do |person|
  expect(person.age).to eq 15
end
```

When

I run `cucumber features/foo.feature` 👍 (012ms)

Then

it should pass 👍 (000ms)

Unicode in tables

You are free to use unicode in your tables: we've taken care to ensure that the tables are properly aligned so that your output is as readable as possible.

tags: @spawn,@spawn

Given

a file named "features/unicode.feature" with: 👍 (000ms)

```
Feature: Featuring unicode

  Scenario: table with unicode
    Given passing
      | Brüno | abc |
      | Bruno | æøå |
```

When

I run `cucumber -q --dry-run features/unicode.feature` 👍 (605ms)

Then

it should pass with: 👍 (000ms)

```
Feature: Featuring unicode

  Scenario: table with unicode
    Given passing
      | Brüno | abc |
      | Bruno | æøå |

1 scenario (1 undefined)
1 step (1 undefined)
```

Usage formatter

In order to see where step definitions are used
Developers should be able to see a list of step definitions and their use

Background

Given

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: F
  Background: A
    Given A
  Scenario: B
    Given B
  Scenario Outline: CA
    Given <x>
    And B
    Examples:
      |x|
      |C|
      |A|
  Scenario: AC
    Given A
    Given C
```

And

a file named "features/step_definitions/steps.rb" with: 👍 (000ms)

```
Given(/A/) { }
Given(/B/) { }
Given(/C/) { }
Given(/D/) { }
```

Run with --format usage

When

I run `cucumber -f usage --dry-run` 👍 (015ms)

Then

it should pass with exactly: 👍 (000ms)

```
-----------

/A/       # features/step_definitions/steps.rb:1
  Given A # features/f.feature:3
  Given A # features/f.feature:12
  Given A # features/f.feature:14
/B/       # features/step_definitions/steps.rb:2
  Given B # features/f.feature:5
  And B   # features/f.feature:11
  And B   # features/f.feature:12
/C/       # features/step_definitions/steps.rb:3
  Given C # features/f.feature:11
  Given C # features/f.feature:15
/D/       # features/step_definitions/steps.rb:4
  NOT MATCHED BY ANY STEPS

4 scenarios (4 skipped)
11 steps (11 skipped)
```

Run with --expand --format usage

When

I run `cucumber -x -f usage --dry-run` 👍 (018ms)

Then

it should pass with exactly: 👍 (000ms)

```
-----------

/A/       # features/step_definitions/steps.rb:1
  Given A # features/f.feature:3
  Given A # features/f.feature:12
  Given A # features/f.feature:14
/B/       # features/step_definitions/steps.rb:2
  Given B # features/f.feature:5
  And B   # features/f.feature:11
  And B   # features/f.feature:12
/C/       # features/step_definitions/steps.rb:3
  Given C # features/f.feature:11
  Given C # features/f.feature:15
/D/       # features/step_definitions/steps.rb:4
  NOT MATCHED BY ANY STEPS

4 scenarios (4 skipped)
11 steps (11 skipped)
```

Run with --format stepdefs

When

I run `cucumber -f stepdefs --dry-run` 👍 (024ms)

Then

it should pass with exactly: 👍 (000ms)

```
-----------

/A/   # features/step_definitions/steps.rb:1
/B/   # features/step_definitions/steps.rb:2
/C/   # features/step_definitions/steps.rb:3
/D/   # features/step_definitions/steps.rb:4
  NOT MATCHED BY ANY STEPS

4 scenarios (4 skipped)
11 steps (11 skipped)
```

Using descriptions to give features context

When writing your feature files its very helpful to use description text at the beginning of the feature file, to write a preamble to the feature describing clearly exactly what the feature does.

You can also write descriptions attached to individual scenarios - see the examples below for how this can be used.

It's possible to have your descriptions run over more than one line, and you can have blank lines too. As long as you don't start a line with a Given, When, Then, Background:, Scenario: or similar, you're fine: otherwise Gherkin will start to pay attention.

Background

Given

the standard step definitions 👍 (000ms)

Everything with a description

Given

a file named "features/test.feature" with: 👍 (000ms)

```
Feature: descriptions everywhere

  We can put a useful description here of the feature, which can
  span multiple lines.

  Background:

    We can also put in descriptions showing what the background is
    doing.

    Given this step passes

  Scenario: I'm a scenario with a description

    You can also put descriptions in front of individual scenarios.

    Given this step passes

  Scenario Outline: I'm a scenario outline with a description

    Scenario outlines can have descriptions.

    Given this step <state>
    Examples: Examples

      Specific examples for an outline are allowed to have
      descriptions, too.

      |  state |
      | passes |
```

When

I run `cucumber -q` 👍 (021ms)

Then

the stderr should not contain anything 👍 (000ms)

Then

it should pass with exactly: 👍 (000ms)

```
Feature: descriptions everywhere

  We can put a useful description here of the feature, which can
  span multiple lines.

  Background:

    We can also put in descriptions showing what the background is
    doing.
    Given this step passes

  Scenario: I'm a scenario with a description

    You can also put descriptions in front of individual scenarios.
    Given this step passes

  Scenario Outline: I'm a scenario outline with a description

    Scenario outlines can have descriptions.
    Given this step <state>

    Examples: Examples

      Specific examples for an outline are allowed to have
      descriptions, too.
      | state  |
      | passes |

2 scenarios (2 passed)
4 steps (4 passed)
```

[[Using-star-notation-instead-of-Given/When/Then, Using star notation instead of Given/When/Then]]
=== **Using star notation instead of Given/When/Then**

Cucumber supports the star notation when writing features: instead of using Given/When/Then, you can simply use a star rather like you would use a bullet point.

When you run the feature for the first time, you still get a nice message showing you the code snippet you need to use to implement the step.

Use some *

Given

a file named "features/f.feature" with: 👍 (000ms)

```
Feature: Star-notation feature
  Scenario: S
    * I have some cukes
```

When

I run `cucumber features/f.feature` 👍 (010ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should pass with: 👍 (000ms)

```
Feature: Star-notation feature

  Scenario: S           # features/f.feature:2
    * I have some cukes # features/f.feature:3

1 scenario (1 undefined)
1 step (1 undefined)
```

And

it should pass with: 👍 (000ms)

```
You can implement step definitions for undefined steps with these snippets:

Given(/^I have some cukes$/) do
  pending # Write code here that turns the phrase above into concrete
actions
end
```

Wire protocol table diffing

In order to use the amazing functionality in the Cucumber table object
As a wire server
I want to be able to ask for a table diff during a step definition invocation

Background

tags: @wire

Given

a file named "features/wired.feature" with: 👍 (000ms)

```
Feature: Hello
  Scenario: Wired
    Given we're all wired
```

And

a file named "features/step_definitions/some_remote_place.wire" with: 👍 (000ms)

```
host: localhost
port: 54321
```

Invoke a step definition tries to diff the table and fails

tags: @wire,@wire,@spawn

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (001ms)

When

I run `cucumber -f progress --backtrace` 👍 (807ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should fail with: 👍 (001ms)

```
F

(::) failed steps (::)

Not same (DifferentException from localhost:54321)
a.cs:12
b.cs:34
features/wired.feature:3:in `Given we're all wired'

Failing Scenarios:
cucumber features/wired.feature:2 # Scenario: Wired

1 scenario (1 failed)
1 step (1 failed)
```

Invoke a step definition tries to diff the table and passes

tags: @wire,@wire

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f progress` 👍 (181ms)

Then

it should pass with: 👍 (001ms)

```
.

1 scenario (1 passed)
1 step (1 passed)
```

Invoke a step definition which successfully diffs a table but then fails

tags: @wire,@wire,@spawn

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f progress` 👍 (908ms)

Then

it should fail with: 👍 (001ms)

```
F

(::) failed steps (::)

I wanted things to be different for us
(Cucumber::WireSupport::WireException)
features/wired.feature:3:in `Given we're all wired'

Failing Scenarios:
cucumber features/wired.feature:2 # Scenario: Wired

1 scenario (1 failed)
1 step (1 failed)
```

Invoke a step definition which asks for an immediate diff that fails

tags: @wire,@wire,@spawn

Given

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f progress` 👍 (808ms)

And

it should fail with exactly: 👍 (001ms)

```
F

(::) failed steps (::)

Tables were not identical:

  | (-) a | (+) b |
 (Cucumber::MultilineArgument::DataTable::Different)
features/wired.feature:3:in `Given we're all wired'

Failing Scenarios:
cucumber features/wired.feature:2 # Scenario: Wired

1 scenario (1 failed)
1 step (1 failed)
0m0.012s
```

Wire protocol tags

In order to use Before and After hooks in a wire server, we send tags with the scenario in the begin_scenario and end_scenario messages

Background

tags: @wire

And

a file named "features/step_definitions/some_remote_place.wire" with: 👍 (000ms)

```
host: localhost
port: 54321
```

Run a scenario

tags: @wire,@wire

Given

a file named "features/wired.feature" with: 👍 (000ms)

```
@foo @bar
Feature: Wired

  @baz
  Scenario: Everybody's Wired
    Given we're all wired
```

And

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f pretty -q` 👍 (145ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should pass with: 👍 (000ms)

```
@foo @bar
Feature: Wired

  @baz
  Scenario: Everybody's Wired
    Given we're all wired

1 scenario (1 passed)
1 step (1 passed)
```

Run a scenario outline example

tags: @wire,@wire

Given

a file named "features/wired.feature" with: 👍 (000ms)

```
@foo @bar
Feature: Wired

  @baz
  Scenario Outline: Everybody's Wired
    Given we're all <something>

  Examples:
    | something |
    | wired     |
```

And

there is a wire server running on port 54321 which understands the following protocol: 👍 (002ms)

When

I run `cucumber -f pretty -q` 👍 (147ms)

Then

the stderr should not contain anything 👍 (001ms)

And

it should pass with: 👍 (001ms)

```
@foo @bar
Feature: Wired

  @baz
  Scenario Outline: Everybody's Wired
    Given we're all <something>

    Examples:
      | something |
      | wired     |

1 scenario (1 passed)
1 step (1 passed)
```

Wire protocol timeouts

We don’t want Cucumber to hang forever on a wire server that’s not even there,
but equally we need to give the user the flexibility to allow step definitions
to take a while to execute, if that’s what they need.

Background

tags: @wire

And

a file named "features/wired.feature" with: 👍 (000ms)

```
Feature: Telegraphy
  Scenario: Wired
    Given we're all wired
```

Try to talk to a server that’s not there

tags: @wire,@wire

Given

a file named "features/step_definitions/some_remote_place.wire" with: 👍 (001ms)

```
host: localhost
port: 54321
```

When

I run `cucumber -f progress` 👍 (012ms)

Then

the stderr should contain: 👍 (000ms)

```
Unable to contact the wire server at localhost:54321
```

Invoke a step definition that takes longer than its timeout

tags: @wire,@wire,@spawn

Given

a file named "features/step_definitions/some_remote_place.wire" with: 👍 (000ms)

```
host: localhost
port: 54321
timeout:
  invoke: 0.1
```

And

there is a wire server on port 54321 which understands the following protocol: 👍 (000ms)

And

the wire server takes 0.2 seconds to respond to the invoke message 👍 (002ms)

When

I run `cucumber -f pretty` 👍 (908ms)

Then

the stderr should not contain anything 👍 (000ms)

And

it should fail with: 👍 (001ms)

```
Feature: Telegraphy

  Scenario: Wired         # features/wired.feature:2
    Given we're all wired # Unknown
      Timed out calling wire server with message 'invoke' (Timeout::Error)
      features/wired.feature:3:in `Given we're all wired'

Failing Scenarios:
cucumber features/wired.feature:2 # Scenario: Wired

1 scenario (1 failed)
1 step (1 failed)
```