Cucumber 3.0 and Beyond

Thomas Haver

tjhaver@gmail.com

Abstract

Cucumber is a tool that supports Behavior Driven Development (BDD), a software development practice that promotes collaboration. Cucumber will execute specifications written in plain language and generate reports to reveal the behavior of software against expectations. In the Fall of 2017, the Cucumber core team formally released Cucumber version 3.0.0, introducing changes that will shift how developers use the tool. Among those changes are Cucumber Expressions, updated Tag Expressions, Retry & Flaky Status, Strict Mode, and a new Events API. This paper will present the new features of Cucumber and how to integrate them into an existing automation suite.

Biography

Thomas is presently serving as a Senior Application Architect for Huntington National Bank. He is responsible for the conversion from manual testing to Ruby/Cucumber automation for the entire enterprise. Originally accountable for the development and maintenance of automation frameworks for Digital Channels, he now leads the automation effort for 73 applications across Huntington that encompass testing, metrics & reporting, data, and environment. Thomas leads the training & technical support for both on-site employees as well as offshore contractors. He has developed test strategies and assisted in coordination between multiple lines to improve delivery effectiveness & resource flexibility.

1 Introduction

The goal of Behavior Driven Development (BDD) is to combine automated acceptance tests, functional requirements, and software documentation into a single format that's understandable by the entire team (North, 2006). The core feature of BDD is collaboration, through which a shared understanding of delivering features to the user is independent of role. In BDD, the "definition of done" is agreed to by the entire team. The development team focuses on making requirements pass in the form of tests. These tests are typically written in a business readable format to emphasize the purpose of creating a shared understanding.

Originally developed by Aslak Hellesoy in 2008 (Hellesoy, 2017), the tool called "Cucumber" can be leveraged to support BDD in creating automated tests, executable specifications, and living documentation. In describing the tool, Aslak wrote, "Cucumber was born out of the frustration with ambiguous requirements and misunderstanding between the people who order the software and those who deliver it" (Hellesoy, 2014). The business readable language for implementing Cucumber scenarios is called Gherkin. Chris Matts is credited with originally inventing the "Given-When-Then" keyword format for examples written as test scenarios (Wynne & Hellesoy, 2012). The Gherkin template implies "Given" some initial context, "When" an event occurs, "Then" ensure some outcome. Following BDD principles, the features written in Gherkin are intended to drive development, not follow it. The top-layer of Cucumber is the Gherkin, and will be the starting point for discussion of this paper.

The development team behind Cucumber implemented changes in five core areas for Cucumber 3.0: (1) Cucumber Expressions; (2) Tag Expressions; (3) Retry & Flaky Status; (4) Strict Mode; and, (5) new Events API (Wynne, 2017). Cucumber Expressions are intended to replace Regular Expressions for matching patterns of user behavior in Gherkin. The Tag Expressions replace prior tag standards to enhance readability. The Retry & Flaky status are intended to identify tests that flicker from pass to fail. The Strict Mode will provide more options with Gherkin that is unimplemented or unfinished. Lastly, the new Events API has been updated to model the execution of automation as a series of distinct events.

2 Cucumber Expressions

Cucumber Expressions are simple patterns for matching Step Definitions with Gherkin steps. The Cucumber development team wanted to replace Regular Expressions with something more user friendly and powerful (Wynne, 2017). Cucumber Expressions are an expression language for matching text and the syntax is altered for legibility. In comparison, Regular Expressions were previously used for more flexibility and control. The development team for Cucumber reasoned that automation developers don't need all the power of Regular Expressions, and that legibility is more important to Gherkin (Wynne, 2017).

Whereas Regular Expressions use capture groups to extract pieces of text, Cucumber Expressions use output parameters (Hellesoy, 2017). An output parameter is simply a name between curly braces. The sample scenario below shows a simple example of using a Cucumber Expression versus earlier versions of Cucumber.

```
@cucumber_expressions_example
Scenario: sample scenario for cucumber expressions using an integer
  Given the user navigates to the Example Search page
  And the user fills in "money" for search field
  When the user selects the 3rd predictive question result
  And the user waits for 2 seconds
  Then the user validates the url of the browser window contains search
```

It's not advisable to use explicit sleeps in test automation, but for the purpose of this example it fits. Prior to Cucumber 3.0, the underlying step definition would be implemented like this:

```
When(/^the user waits for (\d+) seconds$/) do |time|
  sleep time.to_i
end
```

The step would accept one or more digit characters as a parameter, and the sleep method call would wait a certain period of time. With the advent of Cucumber Expressions, the integer is a built-in parameter type. The step definition would now look like this:

```
When("the user waits for {int} second(s)") do |time|
  sleep time
end
```

The above expression passes an Integer from the Gherkin by explicitly calling for an "int" in the step definition. However, the expression would match 2 seconds but not 1.5 seconds. Integers are not the only new parameters added – Float is a valid, built-in parameter type:

```
When("the user waits for {float} second(s)") do |time|
  sleep time
end
```

Now the associated sample scenario will accept fractions of a second because the parameter type was changed to Float.

```
@cucumber_expressions_example
Scenario: sample scenario for cucumber expressions using an integer
  Given the user navigates to the Example Search page
  And the user fills in "money" for search field
  When the user selects the 3rd predictive question result
  And the user waits for 1.5 seconds
  Then the user validates the url of the browser window contains search
```

Another change was made to the step definition example in addition to using a double quoted String in place of "Regexp" and a parameter type. Cucumber Expressions allow for optional text. The step definition, "**And the user waits for 1 seconds**" is grammatically incorrect. Surrounding the text with parentheses makes the text inside optional.

```
When("the user waits for {float} second(s)") do |time|
  sleep time
end
```

```
@cucumber_expressions_example
Scenario: sample scenario for cucumber expressions using an integer
  Given the user navigates to the Example Search page
  And the user fills in "money" for search field
  When the user selects the 3rd predictive question result
  And the user waits for 1 second
  Then the user validates the url of the browser window contains search
```

Before Cucumber 3.0, typically a non-capturing group with optional repetition parameter would be used.

```
When(/^the user waits for (\d+) second(?:s)?$/) do |time|
  sleep time.to_i
end
```

Using an Integer parameter is more appropriate when decimals will never be needed. Cucumber Expressions also allow for alternative text. Sometimes the Gherkin needs two or more options from non-capturing groups to make the Step flow. Instead of using pure regexp to indicate multiple options within a non-capturing group, a forward slash works between two or more options:

Step for Cucumber < 3.0:

```
When(/^the user selects the (\d+)(?:st|nd|rd|th)? (.*) result$/) do |index,
page_element|
```

Step for Cucumber 3.0 and beyond:

```
When("the user selects the {int}st/nd/rd/th {page_element} result") do |index,
page_element|
```

The current built-in parameter types are:

{int}	# for example 71 or -19
{float}	# for example 3.6, .8 or -9.2
{word}	# for example bacon (but not bacon cheeseburger)
{string}	# for example “money" or ‘money'. The quotes are removed from the match.

Custom Parameter Types can be defined to represent types for any Cucumber suite. The benefits are: (1) automatic conversion to custom types; (2) document and evolve ubiquitous domain language (living documentation of variables); and, (3) enforce certain patterns (Wynne, 2017 and Hellesoy, 2017). Start by creating a Parameter Types Ruby file in the Support directory.

Under Parameter Type (reference shown below), “name” is the name of the parameter type, “regexp” (Array format) list of regexps for capture groups, “type” is the return type of the transformed, “transformer” is the lambda that transforms a String to another type, “use_for_snippets” is a Boolean that can be used for snippet generation, and “prefer_for_regexp_match” is a Boolean that should set preference over similar types (Wynne, 2017 and Hellesoy, 2017).

```
ParameterType(
    name: 'name',
    regexp: //,
    type: Type,
    transformer: lambda {|foo| foo.to_type_change},
    use_for_snippets: true or false,
    prefer_for_regexp_match: true or false
)
```

Custom parameter types are new to Cucumber 3.0. In prior versions, regexp for each step was the common practice. Now it’s possible to develop a set of parameters relevant to the application under test, as a form of business logic documentation about application specific criteria. An example is shown below:

```
ParameterType(
    name: 'page_element',
    regexp: /.+/,
    type: String,
    transformer: lambda {|s| s.gsub(/\\"/, "'")},
    use_for_snippets: true,
    prefer_for_regexp_match: false
)
```

The example using "**page_element**" for snippets on web applications is not an ideal implementation. When Cucumber encounters a Gherkin step without a matching Step Definition, it will print a Step Definition snippet with a matching Cucumber Expression as a starting point. Thus, the regexp will match one or more characters, so "page_element" will be recommended for snippet generation everywhere. Therefore, it's important to utilize as specific matchers as possible. Examples would be account numbers of a specific length, dates in a particular format, or word characters that must be phrased in a redundant manner

Regular Expressions will not disappear from Cucumber. A regular expression used with a capture group for an integer, such as **(\d+)**, the captured String will be transformed to an Integer before being passed to the step definition. The Cucumber Expression library's support for Regular Expressions pass capture groups through parameter types' transformers, same as Cucumber Expressions. Cucumber will no longer suggest Regular Expressions in snippets. Cucumber Expressions will instead be recommended.

Step for Cucumber < 3.0:

```
When(/^the user waits for (\d+) second(?:s)?$/) do |time|
  sleep time.to_i
end
```

Step for Cucumber 3.0 and beyond:

```
When(/^the user waits for (\d+) second(?:s)?$/) do |time|
  sleep time
end
```

At the time of this writing, there are multiple unresolved issues with Cucumber Expressions as currently implemented. For one, RubyMine has a difficult time recognizing the Cucumber Expressions, giving the appearance of unimplemented steps when using that particular IDE. Another is when several parameter types share the same Regexp, only one of the parameter types can be preferential. If two parameter types are defined with the same regular expression that are both preferential, an error will be thrown during matching. Execution will have infinite hang-time with an invalid match with Parameter Type from the Gherkin. Typos will ultimately wreck execution. Both "use_for_snippets" and "prefer_for_regexp_match" parameters don't get passed through with the ruby interpreter using RubyMine. Lastly, Flags for regexp area are not recognized; instead, multiline, ignore case, and extended will throw a Cucumber Expression error that the Parameter Type cannot use that option.

3 Tag Expressions

The new Tag Expressions are Boolean expressions of tags with the logical operators "**and**", "**or**", and "**not**". The new Tag Expressions replace the old syntax with a new, plain language readable syntax to specify which scenarios to run. The intention was for Cucumber can store and reuse commonly used cucumber command line arguments for a project in a **cucumber.yml** file (Wynne, 2017). Now, defining a template requires a name and the command-line options to execute with a profile.

Tags are identified within Feature files or Hooks by an “@” symbol. Cucumber uses them as a query language; they help organize features and scenarios, as well as selectively run tests. The new Tag Expressions are used to: (1) run a subset of scenarios and (2) specify a hook that should be executed or excluded for a subset of scenarios (Wynne, 2017).

The command line arguments for execution are stored in the **cucumber.yml** file. Defining a template requires a name and various command-line options to execute with the profile. The execution of a profile requires the **--profile** or **-p** flag. The default profile is named **default** (the profile will execute if a profile is not specified). All common profiles exclude any **Features** or **Scenarios** with the **@wip** or **@manual** tags by using the tilde (**~**) in front of the tag.

```
default: --format pretty --tags ~@wip --tags ~@manual
run: --format pretty --tags @run --tags ~@wip --tags ~@manual
regression: --format pretty --tags @regression --tags ~@wip --tags ~@manual
jenkins: --format json -o cucumber.json --tags ~@wip --tags ~@manual
devops: --format json -o cucumber.json --tags ~@wip --tags ~@manual
```

Cucumber uses tags in multiple ways. With hooks, tags can be added to “**Before**”, “**After**”, “**Around**”, or “**AfterStep**” (Wynne & Hellesoy, 2012). This will execute hooks for only those scenarios with the given tags. A Feature Tag is located above the feature header. Multiple tags can be applied to a feature (separated with a space or newline). A Scenario Tag is located above the scenario header. Multiple tags can be applied to a scenario (separated with a space or newline). Any tag that exists on a Feature will be inherited by each Scenario, Scenario Outline, and Example. The new Tag Expressions for Cucumber 3.0 and beyond will use Boolean expressions of tags with the logical operators “**and**”, “**or**” and “**not**”.

Tag Expressions for Cucumber < 3.0:

```
sample_or_tags_profile: --format pretty --tags @smoke_test,@pbi12345,@pnsqc
sample_and_tags_profile: --format pretty --tags @smoke_test --tags @pbi12345 --
tags @pnsqc
sample_excluding_tags: --format pretty --tags ~@manual --tags @smoke_test
```

Tag Expressions for Cucumber 3.0 and beyond:

```
sample_or_tags_profile: --format pretty --tags '@smoke_test or @pbi12345 or
@pnsqc'
sample_and_tags_profile: --format pretty --tags '@smoke_test and @pbi12345 and
@pnsqc'
sample_excluding_tags: --format pretty --tags '@smoke_test and not @manual'
```

A parentheses can be used for clarity or to change operator precedence:

```
sample_parantheses_profile: --format pretty --tags '@regression and not (@wip or
@manual)'
sample_parantheses_profile: --format pretty --tags '(@pnsqc or @smoke_test) and
(not @manual)'
sample_parantheses_profile: --format pretty --tags '(@smoke_test) and (not
@manual) and (not @wip)'
```

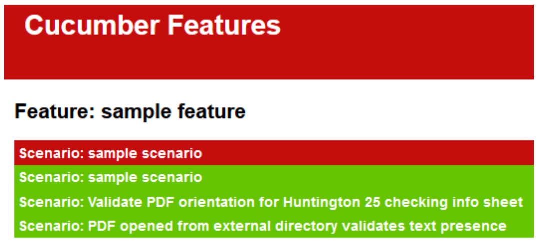
All new Cucumber profiles must use Strings around the tag expression and use logical operators. The optional parentheses are best used for clarity or precedence. Multiple usages of the **--tags** flag in a profile is still allowed. The new syntax also applies to the aforementioned tagged Hooks. There are no changes to how tags are written in Features or Scenarios. There is no change in behavior due to underlying code changes in the Rakefile, Cucumber Profile, or Hooks. There is, however, one warning to note: Cucumber will yield a warning in the console to update the profile usage of tag expressions, but the old way of identifying tags won't be deprecated until **cucumber –v 4.0.0** (Wynne, 2017).

4 Retry and Flaky Status

A Retry flag has been introduced to retry any scenario that fails a given number of times. Scenarios that pass on a subsequent execution are given a "flaky" status. The Cucumber development team wanted to address flickering scenarios, since they reasoned automation developers were implementing their own solution to retry failed scenarios (Wynne, 2017). Retry is now an optional command line argument, which will accept integer values. Any scenario that fails is assigned the "flaky" status upon success after a prior failure. Example execution result is shown below:

```
Flaky Scenarios:
cucumber features/gherkin/sample.feature:3

3 scenarios (1 flaky, 2 passed)
23 steps (1 failed, 2 skipped, 20 passed)
1m1.651s
```

The usage of "retry" is questionable because it already exists as a keyword in Ruby, which can lead to confusion and potential misuse. Also, Flaky status is not present in outputted HTML report or JSON report, but the execution result is present for **Rerun**.


Cucumber Features

Feature: sample feature

Scenario: sample scenario
Scenario: sample scenario
Scenario: Validate PDF orientation for Huntington 25 checking info sheet
Scenario: PDF opened from external directory validates text presence

5 Strict Mode

The updated Strict Mode is more flexible about enforcing rules on the result of Scenarios compared to prior versions of Cucumber. The Cucumber development team changed the original strict mode, which failed if there were any undefined, pending, or failing results. One consequence of the old implementation was any execution in continuous integration (CI) tool such as Jenkins resulted in failures from the execution of many scenarios if just a single issue was present (non-zero exit code). The new Strict Mode implementation is intended to provide more flexibility for execution (Wynne & Hellesoy, 2017). Details of the command options are shown the table below:

Strict Mode Changes	
Command	**Effect**
-S, --[no-]strict	Fail if there are any strict affected results (that is undefined, pending or flaky results).
--[no-]strict-undefined	Fail if there are any undefined results.
--[no-]strict-pending	Fail if there are any pending results.
--[no-]strict-flaky	Fail if there are any flaky results.

A Scenario with an undefined step will have two different exit codes depending on **Strict** mode. If the profile has nothing defined, the exit code is 0. If the profile has **--no-strict-undefined**, the exit code is also 0. If the profile has **--strict-undefined**, the exit code is 1. The console yields a Cucumber exception for that step.

Profile with nothing defined:

```
  Then undefined step

2 scenarios (1 undefined, 1 passed)
5 steps (1 undefined, 7 passed)
0m13.087s

You can implement step definitions for undefined steps with these snippets:

    Then("undefined step") do
      pending # Write code here the phrase here into concrete actions
    end

Process finished with exit code 0
```

Profile with strict undefined set:

```
  Then undefined step
    Undefined step: "undefined step" (Cucumber::Undefined)
    features/gherkin/sample.feature:66:in 'Then undefined step'

Undefined Scenarios:
cucumber features/gherkin/sample.feature:63

2 scenarios (1 undefined, 1 passed)
5 steps (1 undefined, 7 passed)
0m17.919s

You can implement step definitions for undefined steps with these snippets:

Then("undefined step") do
  pending # Write code here the phrase here into concrete actions
end

Process finished with exit code 1
```

A Scenario with a pending step will have two different exit codes depending on **Strict** mode. If the profile has nothing defined, the exit code is 0. If the profile has --**no-strict-pending**, the exit code is also 0. If the profile has --**strict-pending**, the exit code is 1. The console yields a Cucumber exception for that step.

Profile with nothing defined:

```
  Then pending step
    TODO (Cucumber::Pending)
    ./features/step_definitions/custom_steps.rb:19:in '/^pending step$/'
    features/gherkin/sample.feature:66:in 'Then pending step'

2 scenarios (1 pending, 1 passed)
5 steps (1 pending, 7 passed)
0m19.316s

Process finished with exit code 0
```

Profile with strict pending set:

```
  Then pending step
    TODO (Cucumber::Pending)
    ./features/step_definitions/custom_steps.rb:19:in '/^pending step$/'
    features/gherkin/sample.feature:66:in 'Then pending step'

Pending Scenarios:
cucumber features/gherkin/sample.feature:63

2 scenarios (1 pending, 1 passed)
5 steps (1 pending, 7 passed)
0m18.079s

Process finished with exit code 1
```

A Scenario with a flaky step will have two different exit codes depending on **Strict** mode. If the profile has nothing defined, the exit code is 0. If the profile has --**no-strict-flaky**, the exit code is also 0. If the profile has --**strict-flaky**, the exit code is 1.

Profile with nothing defined:

```
Flaky Scenarios:
cucumber features/gherkin/sample.feature:63

2 scenarios (1 flaky, 1 passed)
14 steps (2 failed, 12 passed)
0m36.332s

Process finished with exit code 0
```

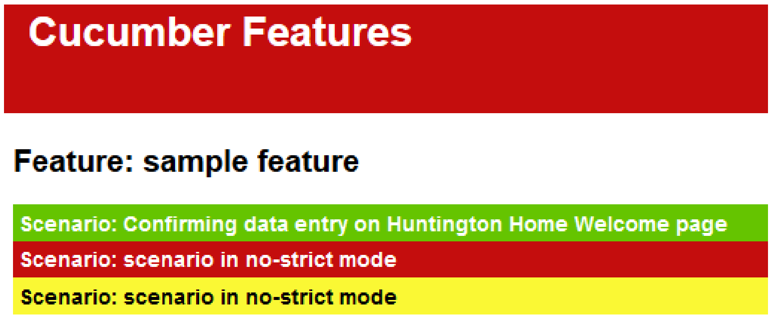
Profile with strict flaky defined:

```
    Then flaky step

Flaky Scenarios:
cucumber features/gherkin/sample.feature:63

2 scenarios (1 flaky, 1 passed)
11 steps (1 failed, 10 passed)
0m21.935s

Process finished with exit code 1
```

A Scenario with all failure types will have two different exit codes depending on **Strict** mode. If the profile has nothing defined, the exit code is 0. If the profile has --**no-strict-** the exit code is also 0 so long as pending or undefined occurs first. Even in **--no-strict** mode, a flaky scenario followed by a pending or undefined step will yield an exit code of 1. If the profile has --**strict**, the exit code is 1 regardless of order.

For any implementation beyond Cucumber 3.0.0, new profiles for strict mode conditions can be created in the **cucumber.yml** file. However, that would require many additional profiles without being able to modify the existing profiles. Instead, any existing profiles prior to version 3.0.0 should remain the same, with optional **strict** parameters. The **strict** flag can be passed in dynamically via environment variable. Without being set, the **strict** environment variable will default to the **--no-strict** flag (the default in Cucumber as before).

Strict Mode will not show you Flaky status either or provide additional context in the output HTML or JSON report. The existing color schemes provided in the formatter will have to be used for guidance. Even in **--no-strict** mode, a flaky scenario followed by a pending or undefined step will yield an exit code of 1; the result should be exit code 0.


Cucumber Features

Feature: sample feature

Scenario: Confirming data entry on Huntington Home Welcome page
Scenario: scenario in no-strict mode
Scenario: scenario in no-strict mode

6 Events API

The Cucumber development team updated the test automation execution of compiling and running features into a series of accessible events. The intent was for events to be read-only, which are used by writing formatters and output tools (such as test result persistence stored in database tables). Many of formatters prior to version 3.0.0 have been rewritten to use the new API, though some still use the old formatter API. The events are described as follows (Wynne, 2017):

- **StepDefinitionRegistered:** “event when the **step_definitions** directory is read”.
- **GherkinSourceRead:** “event when Cucumber reads a **Feature** file”.

- **StepActivated**: “events are fired as each step is activated when the **Feature** file is compiled into test cases”.
- **TestRunStarted:** “event describes what will run after the test cases are compiled”.
- **TestCaseStarted:** “event fires for each test case before any steps run”.
- **TestStepStarted:** “event (including hooks) fire for each Test Step”.
- **TestStepFinished:** “event contains the result and duration of that step”.
- **TestCaseFinished:** “event fires when all the steps have run, and includes the overall result of the test case”.
- **TestRunFinished:** “event fires when all test cases are done”.

There are still unresolved implementations for the new events API. The event, **TestRunFinished**, contains no data. Events API is a work in progress for many formatters. Lastly, the Cucumber Reports Jenkins plugin must be updated to parse Cucumber v. 3.0.0 results.

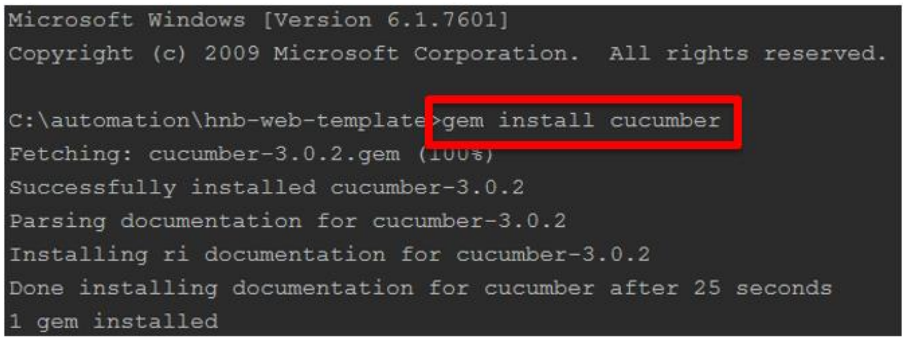
7 Additional Changes Beyond Cucumber 3.0

There are a few other changes to consider:

- Support for Ruby 1.9 and 2.0 has been dropped.
- Transforms have been removed in favor of Parameter Types (Cucumber Expressions).
- The Cucumber Core Team plans to open up contributions to more users in an effort to clean up the code base. The code base will be simplified.
- The old formatter API will be deprecated in Cucumber –v 4.0.0
- Existing Tag Expressions will be supported until Cucumber –v 4.0.0.

8 Appendix: Installation for Ruby with RubyMine IDE

Ruby-Cucumber can be installed via command line directly, bundle installation in the console, or within an IDE such as RubyMine. When the gem version is excluded from the command (e.g., –v 0.0.0), RubyGems will pull the most recent version by default. To install a specific version, include the –v tag in your install command with the specific version number. Alternatively, type "gem install cucumber –v 3.0.0".

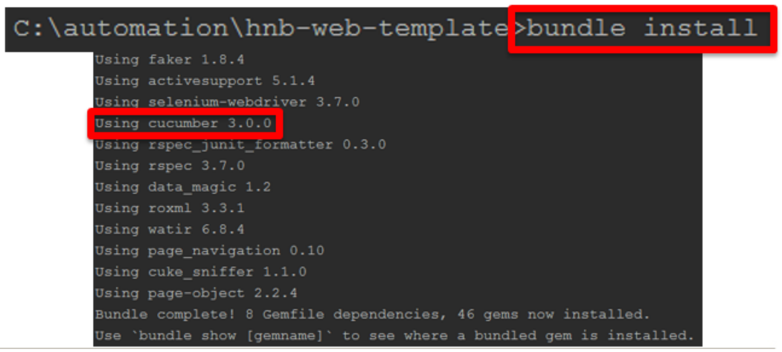


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\automation\hnb-web-template>gem install cucumber
Fetching: cucumber-3.0.2.gem (100%)
Successfully installed cucumber-3.0.2
Parsing documentation for cucumber-3.0.2
Installing ri documentation for cucumber-3.0.2
Done installing documentation for cucumber after 25 seconds
1 gem installed
```

For RubyMine, include 'cucumber' in the Gemfile so the bundle installation will install the other required project gems. If the version number is not included with the gem, bundler will install the most recent version. To update the project suite with PDF Reader, follow these steps in RubyMine: Tools -> Bundler -> Install -> click 'Install'.

```
source 'http://rubygems.org'

group :default do

  gem 'rake'
  gem 'cucumber', '=3.0.0'
  gem 'rspec'
  gem 'page-object'
  gem 'watir'
  gem 'selenium-webdriver'
  gem 'cuke_sniffer'
  gem 'rspec_junit_formatter'
  gem 'pdf-reader'
  gem 'certified'

end
```

For Console Bundle Installation (within the project directory), include 'cucumber' in the Gemfile so the bundle installation will install the other required project gems. If the version number is not included with the gem, bundler will install the most recent version.



```
C:\automation\hnb-web-template>bundle install
Using faker 1.8.4
Using activesupport 5.1.4
Using selenium-webdriver 3.7.0
Using cucumber 3.0.0
Using rspec_junit_formatter 0.3.0
Using rspec 3.7.0
Using data_magic 1.2
Using roxml 3.3.1
Using watir 6.8.4
Using page_navigation 0.10
Using cuke_sniffer 1.1.0
Using page-object 2.2.4
Bundle complete! 8 Gemfile dependencies, 46 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

References

1. North, Dan. 2006 “Introducing BDD.” Dan North & Associates. https://dannorth.net/introducing-bdd/ (accessed November 12, 2017).

2. Hellesoy, Aslak. 2017. “Why is the Cucumber Tool for BDD Named as Such?.” Cucumber Limited. https://www.quora.com/Why-is-the-Cucumber-tool-for-BDD-named-as-such (accessed September 27, 2017).

3. Hellesoy, Aslak. 2014. “The World’s Most Misunderstood Collaboration Tool.” Cucumber Limited. https://cucumber.io/blog/2014/03/03/the-worlds-most-misunderstood-collaboration-tool (accessed September 27, 2017).

4. Wynne, Matt, and Aslak Hellesøy. The Cucumber Book: Behavior-driven Development for Testers and Developers. Dallas, TX: Pragmatic Bookshelf, 2012.

5. Wynne, Matt. 2017. “Announcing Cucumber-Ruby v3.0.0.” Cucumber Limited. https://cucumber.io/blog/2017/09/27/announcing-cucumber-ruby-3-0-0 (accessed September 27, 2017).

6. Hellesoy, Aslak, 2017. “Parameter Types Examples.” Cucumber Limited. https://github.com/cucumber/cucumber-ruby/blob/master/features/docs/writing_support_code/parameter_types.feature (accessed November 12, 2017).

7. Hellesoy, Aslak. 2017. “Announcing Cucumber Expressions.” Cucumber Limited. https://cucumber.io/blog/2017/07/26/announcing-cucumber-expressions (accessed July 26, 2017).

8. Wynne, Matt. 2017. “Cucumber Expressions.” Cucumber Limited. https://docs.cucumber.io/cucumber-expressions/ (accessed November 12, 2017).

9. Wynne, Matt. 2017. “Tag Expressions.” Cucumber Limited. https://docs.cucumber.io/tag-expressions/ (accessed November 12, 2017).

10. Wynne, Matt. 2017. “Retry and Flaky Status.” Cucumber Limited https://app.cucumber.pro/projects/cucumber-ruby/documents/branch/master/features/docs/cli/retry_failing_tests.feature (accessed November 12, 2017).

11. Wynne, Matt, Hellesoy, Aslak & various contributors. 2008-2018. Full Change Log: https://github.com/cucumber/cucumber-ruby/blob/master/CHANGELOG.md (accessed November 12, 2017).

12. Wynne, Matt. 2017. “Events API.” Cucumber Limited. http://www.rubydoc.info/github/cucumber/cucumber-ruby/Cucumber/Events (accessed November 12, 2017).

13. Adzic, Gojko. Specification by Example: How Successful Teams Deliver the Right Software. Shelter Island, NY: Manning, 2011.