

---

# **Behat**

## ***Release 2.5.3***

August 29, 2016



<b>1</b>	<b>Quick Intro</b>	<b>3</b>
1.1	Quick Intro to Behat . . . . .	3
<b>2</b>	<b>Guides</b>	<b>15</b>
2.1	Writing Features - Gherkin Language . . . . .	15
2.2	Defining Reusable Actions - Step Definitions . . . . .	23
2.3	Hooking into the Test Process - Hooks . . . . .	37
2.4	Testing Features - <code>FeatureContext</code> Class . . . . .	43
2.5	Closures as Definitions and Hooks . . . . .	47
2.6	Command Line Tool - <code>behat</code> . . . . .	50
2.7	Configuration - <code>behat.yml</code> . . . . .	57
<b>3</b>	<b>Cookbook</b>	<b>63</b>
3.1	Developing Web Applications with Behat and Mink . . . . .	63
3.2	Migrating from Behat 1.x to 2.0 . . . . .	69
3.3	Using Spin Functions for Slow Tests . . . . .	74
3.4	Using the Symfony2 Profiler with Behat and <code>Symfony2Extension</code> . . . . .	76
3.5	Intercepting the redirection with Behat and Mink . . . . .	78
<b>4</b>	<b>Useful Resources</b>	<b>81</b>
<b>5</b>	<b>More about Behavior Driven Development</b>	<b>83</b>



**Behat v2.x is not maintained anymore. Please upgrade to v3.x ASAP**

Behat is an open source behavior-driven development framework for PHP 5.3 and 5.4. What is *behavior-driven development*, you ask? It's the idea that you start by writing human-readable sentences that describe a feature of your application and how it should work, and only then implement this behavior in software.

For example, imagine you're about to create the famous UNIX `ls` command. Before you begin, you describe how the feature should work:

```
Feature: ls
  In order to see the directory structure
  As a UNIX user
  I need to be able to list the current directory's contents

Scenario:
  Given I am in a directory "test"
  And I have a file named "foo"
  And I have a file named "bar"
  When I run "ls"
  Then I should get:
    """
    bar
    foo
    """
```

As a developer, your work is done as soon as you've made the `ls` command behave as described in the *Scenario*.

Now, wouldn't it be cool if something could read this sentence and use it to actually run a test against the `ls` command? Hey, that's exactly what Behat does! As you'll see, Behat is easy to learn, quick to use, and will put the fun back into your testing.

---

**Note:** Behat was inspired by Ruby's Cucumber project, especially its syntax (called Gherkin).

---



---

## Quick Intro

---

To become *Behat'er* in 20 minutes, just dive into the quick-start guide and enjoy!

### 1.1 Quick Intro to Behat

Welcome to Behat! Behat is a tool that makes [behavior driven development](#) (BDD) possible. With BDD, you write human-readable stories that describe the behavior of your application. These stories can then be auto-tested against your application. And yes, it's as cool as it sounds!

For example, imagine you've been hired to build the famous `ls` UNIX command. A stakeholder may say to you:

**Feature:** `ls`

```
In order to see the directory structure
As a UNIX user
I need to be able to list the current directory's contents
```

**Scenario:** List 2 files in a directory

```
Given I am in a directory "test"
And I have a file named "foo"
And I have a file named "bar"
When I run "ls"
Then I should get:
"""
bar
foo
"""
```

In this tutorial, we'll show you how Behat can execute this simple story as a test that verifies that the `ls` commands works as described.

That's it! Behat can be used to test anything, including web-related behavior via the [Mink](#) library.

---

**Note:** If you want to learn more about the philosophy of testing the “behavior” of your application, see [What's in a Story?](#)

---

---

**Note:** Behat was inspired by Ruby's [Cucumber](#) project.

---

### 1.1.1 Installation

Behat is an executable that you'll run from the command line to execute your stories as tests. Before you begin, ensure that you have at least PHP 5.3.1 installed.

#### Method #1 (Composer)

The simplest way to install Behat is through Composer.

Create `composer.json` file in the project root:

```
{
    "require-dev": {
        "behat/behat": "~2.5"
    },
    "config": {
        "bin-dir": "bin/"
    }
}
```

Then download `composer.phar` and run `install` command:

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar install
```

---

**Note:** Composer uses GitHub zipball service by default and this service is known for outages from time to time. If you get

The ... file could not be downloaded (HTTP/1.1 502 Bad Gateway)

during installation, just use `--prefer-source` option:

```
$ php composer.phar install --prefer-source
```

---

After that, you will be able to run Behat with:

```
$ bin/behat
```

#### Method #2 (PHAR)

Download the PHAR corresponding to the latest 2.5 release on the [Github release page](#).

Now you can execute Behat by simply running phar archive through `php`:

```
$ php behat.phar
```

#### Method #3 (Git)

You can also clone the project with Git by running:

```
$ git clone git://github.com/Behat/Behat.git && cd Behat
```

Then download `composer.phar` and run `install` command:

```
$ wget -nc https://getcomposer.org/composer.phar
$ php composer.phar install
```



After that, you will be able to run Behat with:

```
$ bin/behat
```

## 1.1.2 Basic Usage

In this example, we'll rewind several decades and pretend we're building the original UNIX `ls` command. Create a new directory and setup behat inside that directory:

```
$ mkdir ls_project
$ cd ls_project
$ behat --init
```

The `behat --init` will create a `features/` directory with some basic things to get your started.

### Define your Feature

Everything in Behat always starts with a *feature* that you want to describe and then implement. In this example, the feature will be the `ls` command, which can be thought of as one feature of the whole UNIX system. Since the feature is the `ls` command, start by creating a `features/ls.feature` file:

```
# features/ls.feature
Feature: ls
  In order to see the directory structure
  As a UNIX user
  I need to be able to list the current directory's contents
```

Every feature starts with this same format: a line naming the feature, followed by three lines that describe the benefit, the role and the feature itself. And while this section is required, its contents aren't actually important to Behat or your eventual test. This section is important, however, so that each feature is described consistently and is readable by other people.

### Define a Scenario

Next, add the following scenario to the end of the `features/ls.feature` file:

```
Scenario: List 2 files in a directory
  Given I am in a directory "test"
  And I have a file named "foo"
  And I have a file named "bar"
  When I run "ls"
  Then I should get:
    """
    bar
    foo
    """
```

---

**Tip:** The special `"""` syntax seen on the last few lines is just a special syntax for defining steps on multiple lines. Don't worry about it too much for now.

---

Each feature is defined by one or more “scenarios”, which explain how that feature should act under different conditions. This is the part that will be transformed into a test. Each scenario always follows the same basic format:

```
Scenario: Some description of the scenario
  Given [some context]
  When [some event]
  Then [outcome]
```

Each part of the scenario - the *context*, the *event*, and the *outcome* - can be extended by adding the *And* or *But* keyword:

```
Scenario: Some description of the scenario
  Given [some context]
    And [more context]
  When [some event]
    And [second event occurs]
  Then [outcome]
    And [another outcome]
    But [another outcome]
```

There's no actual difference between *Then*, *And*, *But* or any of the other words that start each line. These keywords are all made available so that your scenarios are natural and readable.

### Executing Behat

You've now defined the feature and one scenario for that feature. You're ready to see Behat in action! Try executing Behat from inside your `ls_project` directory:

```
$ behat
```

If everything worked correctly, you should see something like this:



```

$ behat
Feature: ls
  In order to see the directory structure
  As a UNIX user
  I need to be able to list the current directory's contents

  Scenario: List 2 files in a directory # features/ls.feature:6
    Given I am in a directory "test"
    And I have a file named "foo"
    And I have a file named "bar"
    When I run "ls"
    Then I should get:
      """
      bar
      foo
      """

1 scenario (1 undefined)
5 steps (5 undefined)
0m0.045s

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

/**
 * @Given /^I am in a directory "([^"]*)"$/
 */
public function iAmInADirectory($argument1)
{
    throw new Pending();
}

/**
 * @Given /^I have a file named "([^"]*)"$/
 */
public function iHaveAFileNamed($argument1)
{
    throw new Pending();
}

/**
 * @When /^I run "([^"]*)"$/
 */
public function iRun($argument1)
{
    throw new Pending();
}

/**
 * @Then /^I should get:$/
 */
public function iShouldGet(PyStringNode $string)
{
    throw new Pending();
}

```

## Writing your Step definitions

Behat automatically finds the `features/ls.feature` file and tries to execute its Scenario as a test. However, we haven't told Behat what to do with statements like `Given I am in a directory "test"`, which causes an error. Behat works by matching each statement of a Scenario to a list of regular expression “steps” that you define. In other words, it's your job to tell Behat what to do when it sees `Given I am in a directory "test"`. Fortunately, Behat helps you out by printing the regular expression that you probably need in order to create that step definition:

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

```
/**
 * @Given /^I am in a directory "([^"]*)"$/
 */
public function iAmInADirectory($argument1)
{
    throw new PendingException();
}
```

Let's use Behat's advice and add the following to the `features/bootstrap/FeatureContext.php` file, renaming `$argument1` to `$dir`, simply for clarity:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

class FeatureContext extends BehatContext
{
    /**
     * @Given /^I am in a directory "([^"]*)"$/
     */
    public function iAmInADirectory($dir)
    {
        if (!file_exists($dir)) {
            mkdir($dir);
        }
        chdir($dir);
    }
}
```

Basically, we've started with the regular expression suggested by Behat, which makes the value inside the quotations (e.g. “test”) available as the `$dir` variable. Inside the method, we simply create the directory and move into it.

Repeat this for the other three missing steps so that your `FeatureContext.php` file looks like this:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

class FeatureContext extends BehatContext
{
```

```

private $output;

/** @Given /^I am in a directory "([^"]*)"$/ */
public function iAmInADirectory($dir)
{
    if (!file_exists($dir)) {
        mkdir($dir);
    }
    chdir($dir);
}

/** @Given /^I have a file named "([^"]*)"$/ */
public function iHaveAFileNamed($file)
{
    touch($file);
}

/** @When /^I run "([^"]*)"$/ */
public function iRun($command)
{
    exec($command, $output);
    $this->output = trim(implode("\n", $output));
}

/** @Then /^I should get:$/ */
public function iShouldGet(PyStringNode $string)
{
    if ((string) $string !== $this->output) {
        throw new Exception(
            "Actual output is:\n" . $this->output
        );
    }
}
}

```

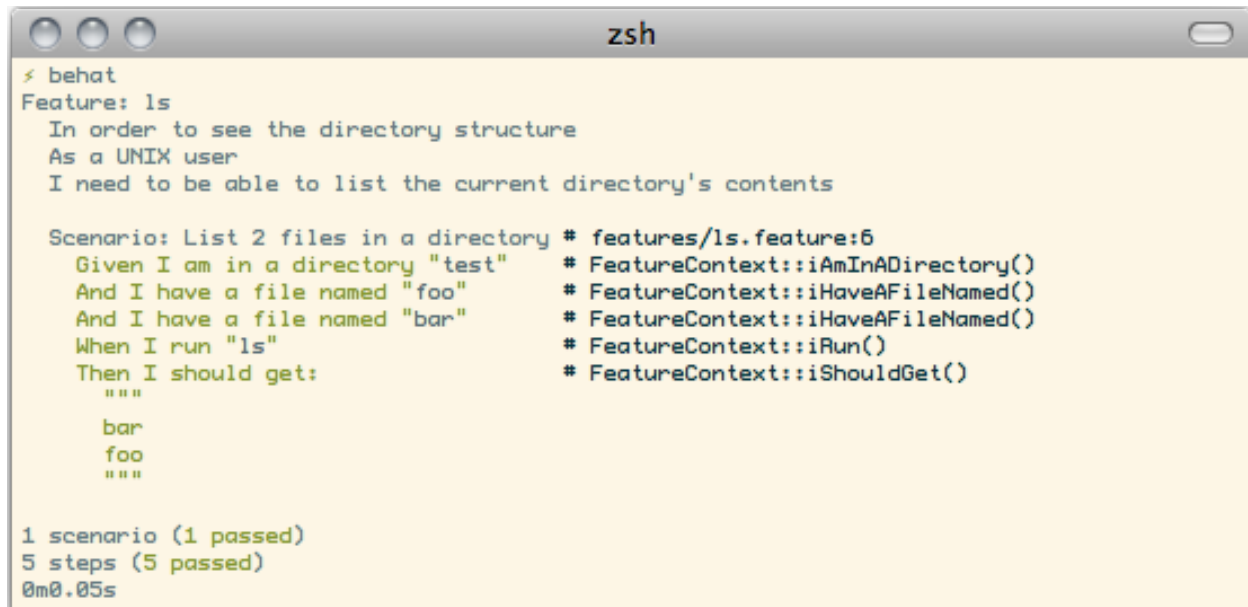
---

**Note:** When you specify multi-line step arguments - like we did using the triple quotation syntax ("""") in the above scenario, the value passed into the step function (e.g. \$string) is actually an object, which can be converted into a string using (string) \$string or \$string->getRaw().

---

Great! Now that you've defined all of your steps, run Behat again:

```
$ behat
```

A screenshot of a zsh terminal window. The window title is 'zsh'. The terminal shows the output of a Behat test run. It starts with a feature definition for 'ls', followed by a scenario 'List 2 files in a directory'. The scenario has five steps: 'Given I am in a directory "test"', 'And I have a file named "foo"', 'And I have a file named "bar"', 'When I run "ls"', and 'Then I should get:'. The 'Then' step has an expected output block containing 'bar' and 'foo'. The test results show '1 scenario (1 passed)' and '5 steps (5 passed)' in green text, with a total time of '0m0.05s'.

```
zsh
< behat
Feature: ls
  In order to see the directory structure
  As a UNIX user
  I need to be able to list the current directory's contents

  Scenario: List 2 files in a directory # features/ls.feature:6
    Given I am in a directory "test" # FeatureContext::iAmInADirectory()
    And I have a file named "foo" # FeatureContext::iHaveAFileNamed()
    And I have a file named "bar" # FeatureContext::iHaveAFileNamed()
    When I run "ls" # FeatureContext::iRun()
    Then I should get: # FeatureContext::iShouldGet()
      """
      bar
      foo
      """

1 scenario (1 passed)
5 steps (5 passed)
0m0.05s
```

Success! Behat executed each of your steps - creating a new directory with two files and running the `ls` command - and compared the result to the expected result.

Of course, now that you've defined your basic steps, adding more scenarios is easy. For example, add the following to your `features/ls.feature` file so that you now have two scenarios defined:

```
Scenario: List 2 files in a directory with the -a option
  Given I am in a directory "test"
  And I have a file named "foo"
  And I have a file named ".bar"
  When I run "ls -a"
  Then I should get:
    """
    .
    ..
    .bar
    foo
    """
```

Run Behat again. This time, it'll run two tests, and both will pass.

```

zsh
$ behat
Feature: ls
  In order to see the directory structure
  As a UNIX user
  I need to be able to list the current directory's contents

  Scenario: List 2 files in a directory # features/ls.feature:6
    Given I am in a directory "test" # FeatureContext::iAmInADirectory()
    And I have a file named "foo" # FeatureContext::iHaveAFileNamed()
    And I have a file named "bar" # FeatureContext::iHaveAFileNamed()
    When I run "ls" # FeatureContext::iRun()
    Then I should get: # FeatureContext::iShouldGet()
    """
    bar
    foo
    """

  Scenario: List 2 files in a directory with the -a option # features/ls.feature:17
    Given I am in a directory "test" # FeatureContext::iAmInADirectory()
    And I have a file named "foo" # FeatureContext::iHaveAFileNamed()
    And I have a file named ".bar" # FeatureContext::iHaveAFileNamed()
    When I run "ls -a" # FeatureContext::iRun()
    Then I should get: # FeatureContext::iShouldGet()
    """
    .
    ..
    .bar
    foo
    """

2 scenarios (2 passed)
10 steps (10 passed)
0m0.541s

```

That's it! Now that you've got a few steps defined, you can probably dream up lots of different scenarios to write for the `ls` command. Of course, this same basic idea could be used to test web applications, and Behat integrates beautifully with a library called [Mink](#) to do just that.

Of course, there's still lot's more to learn, including more about the *Gherkin syntax* (the language used in the `ls.feature` file).

### 1.1.3 Some more Behat Basics

When you run `behat --init`, it sets up a directory that looks like this:

```

|-- features
    |-- bootstrap
        |-- FeatureContext.php

```

Everything related to Behat will live inside the `features` directory, which is composed of three basic areas:

1. `features/` - Behat looks for `*.feature` files here to execute
2. `features/bootstrap/` - Every `*.php` file in that directory will be autoloaded by Behat before any actual steps are executed
3. `features/bootstrap/FeatureContext.php` - This file is the context class in which every scenario step will be executed

### 1.1.4 More about Features

As you've already seen, a feature is a simple, readable plain text file, in a format called Gherkin. Each feature file follows a few basic rules:

1. Every `*.feature` file conventionally consists of a single “feature” (like the `ls` command or *user registration*).
2. A line starting with the keyword `Feature:` followed by its title and three indented lines defines the start of a new feature.
3. A feature usually contains a list of scenarios. You can write whatever you want up until the first scenario: this text will become the feature description.
4. Each scenario starts with the `Scenario:` keyword followed by a short description of the scenario. Under each scenario is a list of steps, which must start with one of the following keywords: `Given`, `When`, `Then`, `But` or `And`. Behat treats each of these keywords the same, but you should use them as intended for consistent scenarios.

---

**Tip:** Behat also allows you to write your features in your native language. In other words, instead of writing `Feature`, `Scenario` or `Given`, you can use your native language by configuring Behat to use one of its many supported languages.

To check if your language is supported and to see the available keywords, run:

```
$ behat --story-syntax --lang YOUR_LANG
```

Supported languages include (but are not limited to) `fr`, `es`, `it` and, of course, the english pirate dialect `en-pirate`.

Keep in mind, that any language, different from `en` should be explicitly marked with `# language: ... comment` at the beginning of your `*.feature` file:

```
# language: fr
Fonctionnalité: ...
...
```

---

You can read more about features and Gherkin language in “*Writing Features - Gherkin Language*” guide.

### 1.1.5 More about Steps

For each step (e.g. `Given I am in a directory "test"`), Behat will look for a matching step definition by matching the text of the step against the regular expressions defined by each step definition.

A step definition is written in php and consists of a keyword, a regular expression, and a callback. For example:

```
/**
 * @Given /^I am in a directory "([^"]*)"$/
 */
public function iAmInADirectory($dir)
{
    if (!file_exists($dir)) {
        mkdir($dir);
    }
    chdir($dir);
}
```

A few pointers:



1. @Given is a definition keyword. There are 3 supported keywords in annotations: @Given/@When/@Then. These three definition keywords are actually equivalent, but all three are available so that your step definition remains readable.
2. The text after the keyword is the regular expression (e.g. /^I am in a directory "([^"]\*)"\$/).
3. All search patterns in the regular expression (e.g. ([^"]\*)) will become method arguments (\$dir).
4. If, inside a step, you need to tell Behat that some sort of “failure” has occurred, you should throw an exception:

```
/**
 * @Then /^I should get:$/
 */
public function iShouldGet(PyStringNode $string)
{
    if ((string) $string !== $this->output) {
        throw new Exception(
            "Actual output is:\n" . $this->output
        );
    }
}
```

**Tip:** Behat doesn't come with its own assertion tool, but you can use any proper assertion tool out there. Proper assertion tool is a library, which assertions throw exceptions on fail. For example, if you're familiar with PHPUnit, you can use its assertions in Behat:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext;
use Behat\Gherkin\Node\PyStringNode;

require_once 'PHPUnit/Autoload.php';
require_once 'PHPUnit/Framework/Assert/Functions.php';

class FeatureContext extends BehatContext
{
    /**
     * @Then /^I should get:$/
     */
    public function iShouldGet(PyStringNode $string)
    {
        assertEquals($string->getRaw(), $this->output);
    }
}
```

In the same way, any step that does *not* throw an exception will be seen by Behat as “passing”.

You can read more about step definitions in “*Defining Reusable Actions - Step Definitions*” guide.

### 1.1.6 The Context Class: FeatureContext

Behat creates a context object for each scenario and executes all scenario steps inside that same object. In other words, if you want to share variables between steps, you can easily do that by setting property values on the context object itself (which was shown in the previous example).

You can read more about FeatureContext in “*Testing Features - FeatureContext Class*” guide.

### 1.1.7 The `behat` Command Line Tool

Behat comes with a powerful console utility responsible for executing the Behat tests. The utility comes with a wide array of options.

To see options and usage for the utility, run:

```
$ behat -h
```

One of the handiest things it does is to show you all of the step definitions that you have configured in your system. This is an easy way to recall exactly how a step you defined earlier is worded:

```
$ behat -dl
```

You can read more about Behat CLI in “*Command Line Tool - behat*” guide.

### 1.1.8 What’s Next?

Congratulations! You now know everything you need in order to get started with behavior driven development and Behat. From here, you can learn more about the *Gherkin* syntax or learn how to test your web applications by using Behat with Mink.

- *Developing Web Applications with Behat and Mink*
- *Writing Features - Gherkin Language*
- *Command Line Tool - behat*

Learn Behat with the topical guides:

## 2.1 Writing Features - Gherkin Language

Behat is a tool to test the behavior of your application, described in special language called Gherkin. Gherkin is a [Business Readable, Domain Specific Language](#) created especially for behavior descriptions. It gives you the ability to remove logic details from behavior tests.

Gherkin serves two purposes: serving as your project's documentation and automated tests. Behat also has a bonus feature: it talks back to you using real, human language telling you what code you should write.

If you're still new to Behat, jump into the [Quick Intro to Behat](#) first, then return here to learn more about Gherkin.

### 2.1.1 Gherkin Syntax

Like YAML or Python, Gherkin is a line-oriented language that uses indentation to define structure. Line endings terminate statements (called steps) and either spaces or tabs may be used for indentation. (We suggest you use spaces for portability.) Finally, most lines in Gherkin start with a special keyword:

```
Feature: Some terse yet descriptive text of what is desired
  In order to realize a named business value
  As an explicit system actor
  I want to gain some beneficial outcome which furthers the goal
```

```
Scenario: Some determinable business situation
  Given some precondition
    And some other precondition
  When some action by the actor
    And some other action
    And yet another action
  Then some testable outcome is achieved
    And something else we can check happens too
```

```
Scenario: A different situation
  ...
```

The parser divides the input into features, scenarios and steps. Let's walk through the above example:

1. **Feature:** Some terse yet descriptive text of what is desired starts the feature and gives it a title. Learn more about features in the [“Features”](#) section.

2. Behat does not parse the next 3 lines of text. (In order to... As an... I want to...). These lines simply provide context to the people reading your feature, and describe the business value derived from the inclusion of the feature in your software.
3. **Scenario:** Some determinable business situation starts the scenario, and contains a description of the scenario. Learn more about scenarios in the “[Scenarios](#)” section.
4. The next 7 lines are the scenario steps, each of which is matched to a regular expression defined elsewhere. Learn more about steps in the “[Steps](#)” section.
5. **Scenario:** A different situation starts the next scenario, and so on.

When you’re executing the feature, the trailing portion of each step (after keywords like *Given*, *And*, *When*, etc) is matched to a regular expression, which executes a PHP callback function. You can read more about steps matching and execution in *Defining Reusable Actions - Step Definitions*.

## 2.1.2 Features

Every \*.feature file conventionally consists of a single feature. Lines starting with the keyword **Feature:** (or its localized equivalent) followed by three indented lines starts a feature. A feature usually contains a list of scenarios. You can write whatever you want up until the first scenario, which starts with **Scenario:** (or localized equivalent) on a new line. You can use [tags](#) to group features and scenarios together, independent of your file and directory structure.

Every scenario consists of a list of [steps](#), which must start with one of the keywords *Given*, *When*, *Then*, *But* or *And* (or localized one). Behat treats them all the same, but you shouldn’t. Here is an example:

```
Feature: Serve coffee
  In order to earn money
  Customers should be able to
  buy coffee at all times

Scenario: Buy last coffee
  Given there are 1 coffees left in the machine
  And I have deposited 1 dollar
  When I press the coffee button
  Then I should be served a coffee
```

In addition to basic [scenarios](#), feature may contain [scenario outlines](#) and [backgrounds](#).

## 2.1.3 Scenarios

Scenario is one of the core Gherkin structures. Every scenario starts with the **Scenario:** keyword (or localized one), followed by an optional scenario title. Each feature can have one or more scenarios, and every scenario consists of one or more [steps](#).

The following scenarios each have 3 steps:

```
Scenario: Wilson posts to his own blog
  Given I am logged in as Wilson
  When I try to post to "Expensive Therapy"
  Then I should see "Your article was published."

Scenario: Wilson fails to post to somebody else’s blog
  Given I am logged in as Wilson
  When I try to post to "Greg’s anti-tax rants"
  Then I should see "Hey! That’s not your blog!"
```

```
Scenario: Greg posts to a client's blog
  Given I am logged in as Greg
  When I try to post to "Expensive Therapy"
  Then I should see "Your article was published."
```

## 2.1.4 Scenario Outlines

Copying and pasting scenarios to use different values can quickly become tedious and repetitive:

```
Scenario: Eat 5 out of 12
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers
```

```
Scenario: Eat 5 out of 20
  Given there are 20 cucumbers
  When I eat 5 cucumbers
  Then I should have 15 cucumbers
```

Scenario Outlines allow us to more concisely express these examples through the use of a template with placeholders:

```
Scenario Outline: Eating
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers
```

### Examples:

start	eat	left
12	5	7
20	5	15

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 of column headers).

The Scenario Outline uses placeholders, which are contained within < > in the Scenario Outline's steps. For example:

```
Given <I'm a placeholder and I'm ok>
```

Think of a placeholder like a variable. It is replaced with a real value from the Examples: table row, where the text between the placeholder angle brackets matches that of the table column header. The value substituted for the placeholder changes with each subsequent run of the Scenario Outline, until the end of the Examples table is reached.

---

**Tip:** You can also use placeholders in [Multiline Arguments](#).

---

**Note:** Your step definitions will never have to match the placeholder text itself, but rather the values replacing the placeholder.

---

So when running the first row of our example:

```
Scenario Outline: controlling order
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers
```

### Examples:

```
| start | eat | left |  
| 12    | 5   | 7    |
```

The scenario that is actually run is:

```
Scenario Outline: controlling order  
# <start> replaced with 12:  
Given there are 12 cucumbers  
# <eat> replaced with 5:  
When I eat 5 cucumbers  
# <left> replaced with 7:  
Then I should have 7 cucumbers
```

## 2.1.5 Backgrounds

Backgrounds allows you to add some context to all scenarios in a single feature. A Background is like an untitled scenario, containing a number of steps. The difference is when it is run: the background is run before each of your scenarios, but after your BeforeScenario hooks (*Hooking into the Test Process - Hooks*).

**Feature:** Multiple site support

**Background:**

```
Given a global administrator named "Greg"  
And a blog named "Greg's anti-tax rants"  
And a customer named "Wilson"  
And a blog named "Expensive Therapy" owned by "Wilson"
```

**Scenario:** Wilson posts to his own blog

```
Given I am logged in as Wilson  
When I try to post to "Expensive Therapy"  
Then I should see "Your article was published."
```

**Scenario:** Greg posts to a client's blog

```
Given I am logged in as Greg  
When I try to post to "Expensive Therapy"  
Then I should see "Your article was published."
```

## 2.1.6 Steps

Features consist of steps, also known as **Givens**, **Whens** and **Thens**.

Behat doesn't technically distinguish between these three kind of steps. However, we strongly recommend that you do! These words have been carefully selected for their purpose, and you should know what the purpose is to get into the BDD mindset.

Robert C. Martin has written a [great post](#) about BDD's Given-When-Then concept where he thinks of them as a finite state machine.

### Givens

The purpose of **Given** steps is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the **When** steps). Avoid talking about user interaction in givens. If you have worked with use cases, givens are your preconditions.

---

**Note:** Two good examples of using **Givens** are:

- To create records (model instances) or set up the database:

```
Given there are no users on site
Given the database is clean
```

- Authenticate a user (An exception to the no-interaction recommendation. Things that “happened earlier” are ok):

```
Given I am logged in as "Everzet"
```

---

**Tip:** It’s ok to call into the layer “inside” the UI layer here (in symfony: talk to the models).

---

And for all the symfony users out there, we recommend using a Given step with a `tables` arguments to set up records instead of fixtures. This way you can read the scenario all in one place and make sense out of it without having to jump between files:

```
Given there are users:
| username | password | email |
| everzet  | 123456   | everzet@knplabs.com |
| fabpot   | 22@222   | fabpot@symfony.com  |
```

## Whens

The purpose of **When** steps is to **describe the key action** the user performs (or, using Robert C. Martin’s metaphor, the state transition).

---

**Note:** Two good examples of **Whens** use are:

- Interact with a web page (the Mink library gives you many web-friendly When steps out of the box):

```
When I am on "/some/page"
When I fill "username" with "everzet"
When I fill "password" with "123456"
When I press "login"
```

- Interact with some CLI library (call commands and record output):

```
When I call "ls -la"
```

## Thens

The purpose of **Then** steps is to **observe outcomes**. The observations should be related to the business value/benefit in your feature description. The observations should inspect the output of the system (a report, user interface, message, command output) and not something deeply buried inside it (that has no business value and is instead part of the implementation).

- Verify that something related to the Given+When is (or is not) in the output
- Check that some external system has received the expected message (was an email with specific content successfully sent?)

```
When I call "echo hello"
Then the output should be "hello"
```

**Note:** While it might be tempting to implement Then steps to just look in the database – resist the temptation. You should only verify output that is observable by the user (or external system). Database data itself is only visible internally to your application, but is then finally exposed by the output of your system in a web browser, on the command-line or an email message.

---

### And, But

If you have several Given, When or Then steps you can write:

```
Scenario: Multiple Givens
  Given one thing
  Given an other thing
  Given yet an other thing
  When I open my eyes
  Then I see something
  Then I don't see something else
```

Or you can use **And** or **But** steps, allowing your Scenario to read more fluently:

```
Scenario: Multiple Givens
  Given one thing
  And an other thing
  And yet an other thing
  When I open my eyes
  Then I see something
  But I don't see something else
```

If you prefer, you can indent scenario steps in a more *programmatic* way, much in the same way your actual code is indented to provide visual context:

```
Scenario: Multiple Givens
  Given one thing
    And an other thing
    And yet an other thing
  When I open my eyes
  Then I see something
    But I don't see something else
```

Behat interprets steps beginning with And or But exactly the same as all other steps. It doesn't differ between them - you should!

## 2.1.7 Multiline Arguments

The regular expression matching in [steps](#) lets you capture small strings from your steps and receive them in your step definitions. However, there are times when you want to pass a richer data structure from a step to a step definition.

This is what multiline step arguments are for. They are written on lines immediately following a step, and are passed to the step definition method as the last argument.

Multiline step arguments come in two flavours: [tables](#) or [pystrings](#).

### Tables

Tables as arguments to steps are handy for specifying a larger data set - usually as input to a Given or as expected output from a Then.



**Scenario:****Given** the following people exist:

name	email	phone
Aslak	aslak@email.com	123
Joe	joe@email.com	234
Bryan	bryan@email.org	456

**Note:** Don't be confused with tables from [scenario outlines](#) - syntactically they are identical, but have a different purpose.

**Tip:** A matching definition for this step looks like this:

```
/**
 * @Given /the following people exist:/
 */
public function thePeopleExist(TableNode $table)
{
    $hash = $table->getHash();
    foreach ($hash as $row) {
        // $row['name'], $row['email'], $row['phone']
    }
}
```

**Note:** A table is injected into a definition as a `TableNode` object, from which you can get hash by columns (`TableNode::getHash()` method) or by rows (`TableNode::getRowsHash()`).

## PyStrings

Multiline Strings (also known as PyStrings) are handy for specifying a larger piece of text. This is done using the so-called PyString syntax. The text should be offset by delimiters consisting of three double-quote marks (""" ) on lines by themselves:

**Scenario:****Given** a blog post named "Random" with:

```
"""
Some Title, Eh?
=====
Here is the first paragraph of my blog post.
Lorem ipsum dolor sit amet, consectetur adipiscing
elit.
"""
```

**Note:** The inspiration for PyString comes from Python where """ is used to delineate docstrings, much in the way /\* ... \*/ is used for multiline docblocks in PHP.

**Tip:** In your step definition, there's no need to find this text and match it in your regular expression. The text will automatically be passed as the last argument into the step definition method. For example:

```
/**
 * @Given /a blog post named "([^"]+)" with:/
 */
public function blogPost($title, PyStringNode $markdown)
{
```

```
$this->createPost($title, $markdown->getRaw());  
}
```

---

**Note:** PyStrings are stored in a `PyStringNode` instance, which you can simply convert to a string with `(string)$pystring` or `$pystring->getRaw()` as in the example above.

---

**Note:** Indentation of the opening `"""` is not important, although common practice is two spaces in from the enclosing step. The indentation inside the triple quotes, however, is significant. Each line of the string passed to the step definition's callback will be de-indented according to the opening `"""`. Indentation beyond the column of the opening `"""` will therefore be preserved.

---

## 2.1.8 Tags

Tags are a great way to organize your features and scenarios. Consider this example:

```
@billing  
Feature: Verify billing  
  
    @important  
    Scenario: Missing product description  
  
    Scenario: Several products
```

A Scenario or Feature can have as many tags as you like, just separate them with spaces:

```
@billing @bicker @annoy  
Feature: Verify billing
```

**Note:** If a tag exists on a Feature, Behat will assign that tag to all child Scenarios and Scenario Outlines too.

---

## 2.1.9 Gherkin in Many Languages

Gherkin is available in many languages, allowing you to write stories using localized keywords from your language. In other words, if you speak French, you can use the word `Fonctionnalité` instead of `Feature`.

To check if Behat and Gherkin support your language (for example, French), run:

```
behat --story-syntax --lang=fr
```

---

**Note:** Keep in mind that any language different from `en` should be explicitly marked with a `# language: ...` comment at the beginning of your `*.feature` file:

```
# language: fr  
Fonctionnalité: ...  
...
```

This way your features will hold all the information about its content type, which is very important for methodologies like BDD, and will also give Behat the ability to have multilanguage features in one suite.

---

## 2.2 Defining Reusable Actions - Step Definitions

*Gherkin language* provides a way to describe your application behavior in business readable language. But how do you test that the described behavior is actually implemented? Or that the application satisfies our business expectations described in feature scenarios? Behat provides a way to map 1-to-1 your scenario steps (actions) to actual PHP code called step definitions:

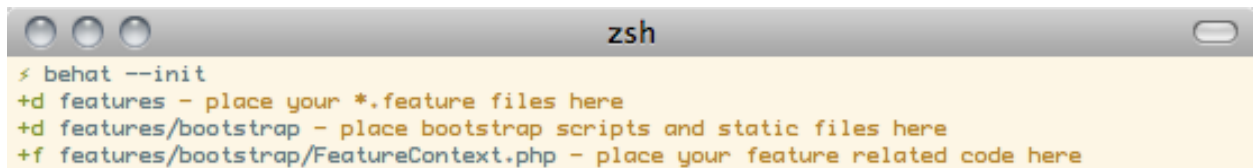
```
/**
 * @When /^I do something with "([^"]*)"$/
 */
public function iDoSomethingWith($argument)
{
    // do something with $argument
}
```

### 2.2.1 Definitions Home - FeatureContext Class

Step definitions are just normal PHP methods. They are instance methods in a special class called *Feature Context*. This class can be easily created by running behat with the `--init` option from your project's directory:

```
$ behat --init
```

After you run this command, Behat will set up a `features` directory inside your project:



```
zsh
$ behat --init
+ d features - place your *.feature files here
+ d features/bootstrap - place bootstrap scripts and static files here
+ f features/bootstrap/FeatureContext.php - place your feature related code here
```

The newly created `features/bootstrap/FeatureContext.php` will have an initial context class to get you started:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\TranslatedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

class FeatureContext extends BehatContext
{
    // Your definitions will live here
}
```

All step definitions and *hooks* necessary for behavior testing your project will be represented as methods inside this class.

### 2.2.2 Creating Your First Step Definition

The main goal for step definitions is to be executed when its matching step is run in Behat. But just because a method exists within `FeatureContext` doesn't mean Behat can find it. Behat needs a way to check that a concrete class

method is suitable for a concrete step in a scenario. Behat matches `FeatureContext` methods to step definitions using regular expression matching.

When Behat runs, it compares special lines of Gherkin from each scenario to the regular expressions bound to each method in `FeatureContext`. If the line of Gherkin satisfies a bound regular expression, its corresponding step definition is executed. It's that simple!

Behat uses phpDoc annotations to bind regular expressions to `FeatureContext` methods:

```
/**
 * @When /^I do something with "([^"]*)"$/
 */
public function someMethod($methodArgument) {}
```

Let's take a closer look at this code:

1. `@When` is a definition keyword. There are 3 supported keywords in annotations: `@Given/@When/@Then`. These three definition keywords are actually equivalent, but all three are available so that your step definition remains readable.
2. The text after the keyword is the regular expression (e.g. `/^I am in a directory "([^"]*)"$/`).
3. All search patterns in the regular expression (e.g. `(["^"]*)`) will become method arguments (`$methodArgument`).

---

**Note:** Notice the comment block starts with `/**`, and not the usual `/*`. This is important for Behat to be able to parse such comments as annotations!

---

As you probably noticed, this regular expression is quite general and its corresponding method will be called for steps that contain ... I do something with "...", including:

```
Given I do something with "string1"
When I do something with "some other string"
Then I do something with "smile :-)"
```

The only real difference between those steps in the eyes of Behat is the text inside double quotes. This text will be passed to its step's corresponding method as an argument value. In the example above, `FeatureContext::someMethod()` will be called three times, each time with a different argument:

1. `->someMethod( $methodArgument = 'string1' );`.
2. `->someMethod( $methodArgument = 'some other string' );`.
3. `->someMethod( $methodArgument = 'smile :-)' );`.

---

**Note:** Regular expression can't automatically determine the datatype of its matches. So all method arguments coming from step definitions are passed as strings. Even if your regular expression matches "500", which could be considered an integer, "500" will be passed as a string argument to the step definition's method.

This is not a feature or limitation of Behat, but rather the inherent way regular expression matching works. It is your responsibility to cast string arguments to integers, floats or booleans where applicable given the code you are testing.

Casting arguments to specific types can be accomplished using [step argument transformations](#).

---

**Note:** Behat does not differentiate between step keywords when matching regular expressions to methods. So a step defined with `@When` could also be matched to `@Given ...`, `@Then ...`, `@And ...`, `@But ...`, etc.

---

Your step definitions can also define multiple arguments to pass to its matching `FeatureContext` method:

```
/**
 * @When /^I do something with "([^"]*)" and with (\d+)$/
 */
public function someMethod($stringArgument, $numberArgument) {}
```

### 2.2.3 Definition Snippets

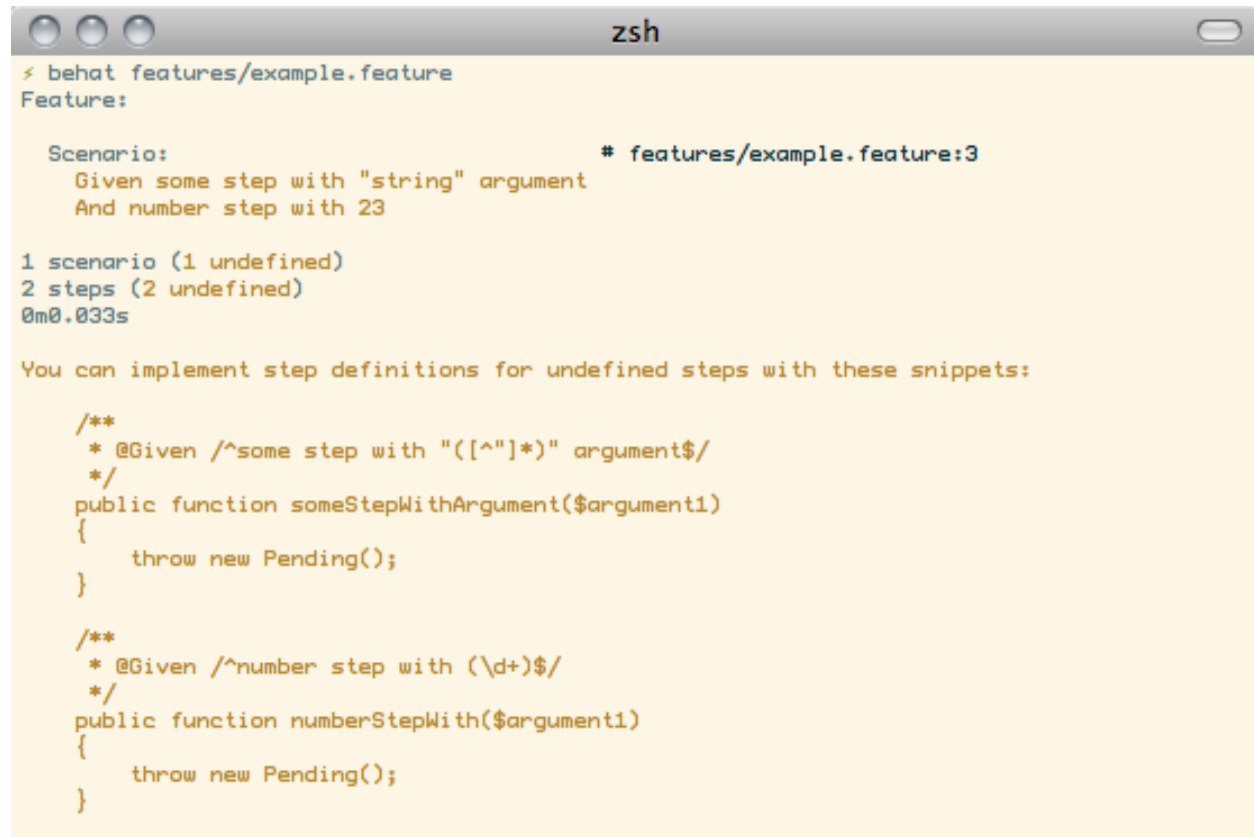
You now know how to write step definitions by hand, but writing all these method stubs, annotations and regular expressions by hands is tedious. Behat makes this routine task much easier and fun by generating definition snippets for you! Let's pretend that you have this feature:

```
# features/example.feature
Feature:
  Scenario:
    Given some step with "string" argument
    And number step with 23
```

Run this feature in Behat:

```
$ behat features/example.feature
```

Behat will provide auto-generated snippets for your steps:



```
zsh
$ behat features/example.feature
Feature:

  Scenario:                                     # features/example.feature:3
    Given some step with "string" argument
    And number step with 23

1 scenario (1 undefined)
2 steps (2 undefined)
0m0.033s

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

/**
 * @Given /^some step with "([^"]*)" argument$/
 */
public function someStepWithArgument($argument1)
{
    throw new Pending();
}

/**
 * @Given /^number step with (\d+)$/
 */
public function numberStepWith($argument1)
{
    throw new Pending();
}
```

It not only generates the proper definition annotation type (@Given), but also a regular expression with string ("([^"]+)") or number ((\d+)) capturing, method name (someStepWithArgument()), numberStepWith()) and arguments (\$argument1), all based just on text of the step. Isn't that cool?

The only thing left for you to do is to copy that method snippets into your `FeatureContext` class and provide a useful body for them!

## 2.2.4 Step Execution Result Types

Now you know how to map actual code to PHP code that will be executed. But how can you tell what exactly “failed” or “passed” when executing a step? And how does Behat actually check that a step executed properly?

For that we have step execution types. Behat differentiates between seven types of step execution results: “Successful Steps”, “Undefined Steps”, “Pending Steps”, “Failed Steps”, “Skipped Steps”, “Ambiguous Steps” and “Redundant Step Definitions”.

Let’s use our previously introduced feature for all the following examples:

```
# features/example.feature
Feature:
  Scenario:
    Given some step with "string" argument
    And number step with 23
```

### Successful Steps

When Behat finds a matching step definition it will execute it. If the definition method does **not** throw an `Exception`, the step is marked as successful (green). What you return from a definition method has no significance on the passing or failing status of the definition itself.

Let’s pretend our context class contains the code below:

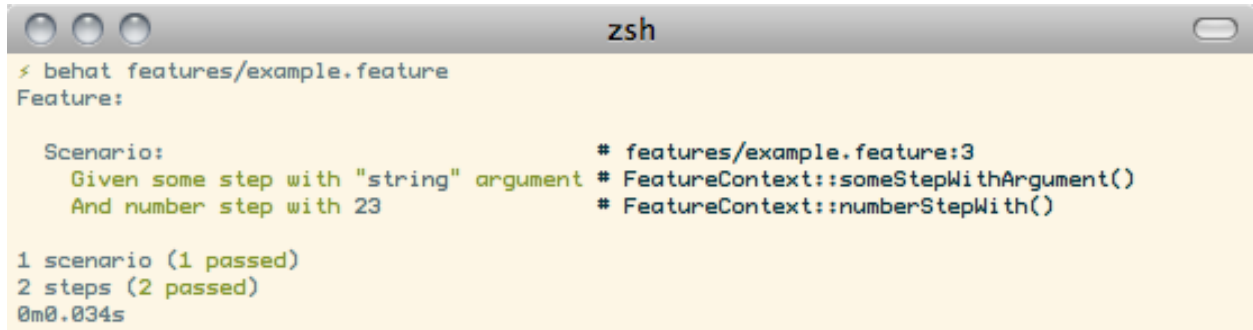
```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext;

class FeatureContext extends BehatContext
{
    /** @Given /^some step with "([^"]*)" argument$/ */
    public function someStepWithArgument($argument1)
    {
    }

    /** @Given /^number step with (\d+)$/ */
    public function numberStepWith($argument1)
    {
    }
}
```

When you run your feature, you’ll see all steps passed and are marked green:



```

zsh
$ behat features/example.feature
Feature:

    Scenario:                                     # features/example.feature:3
        Given some step with "string" argument # FeatureContext::someStepWithArgument()
        And number step with 23                 # FeatureContext::numberStepWith()

1 scenario (1 passed)
2 steps (2 passed)
0m0.034s

```

---

**Note:** Passed steps are always marked as **green** if colors are supported by your console.

---



---

**Tip:** Install `php5-posix` on Linux, Mac OS or other Unix system to be able to see colorful Behat output.

---

## Undefined Steps

When Behat cannot find a matching definition, the step are marked as **undefined**, and all subsequent steps in the scenario are **skipped**.

Let's pretend we have an empty context class:

```

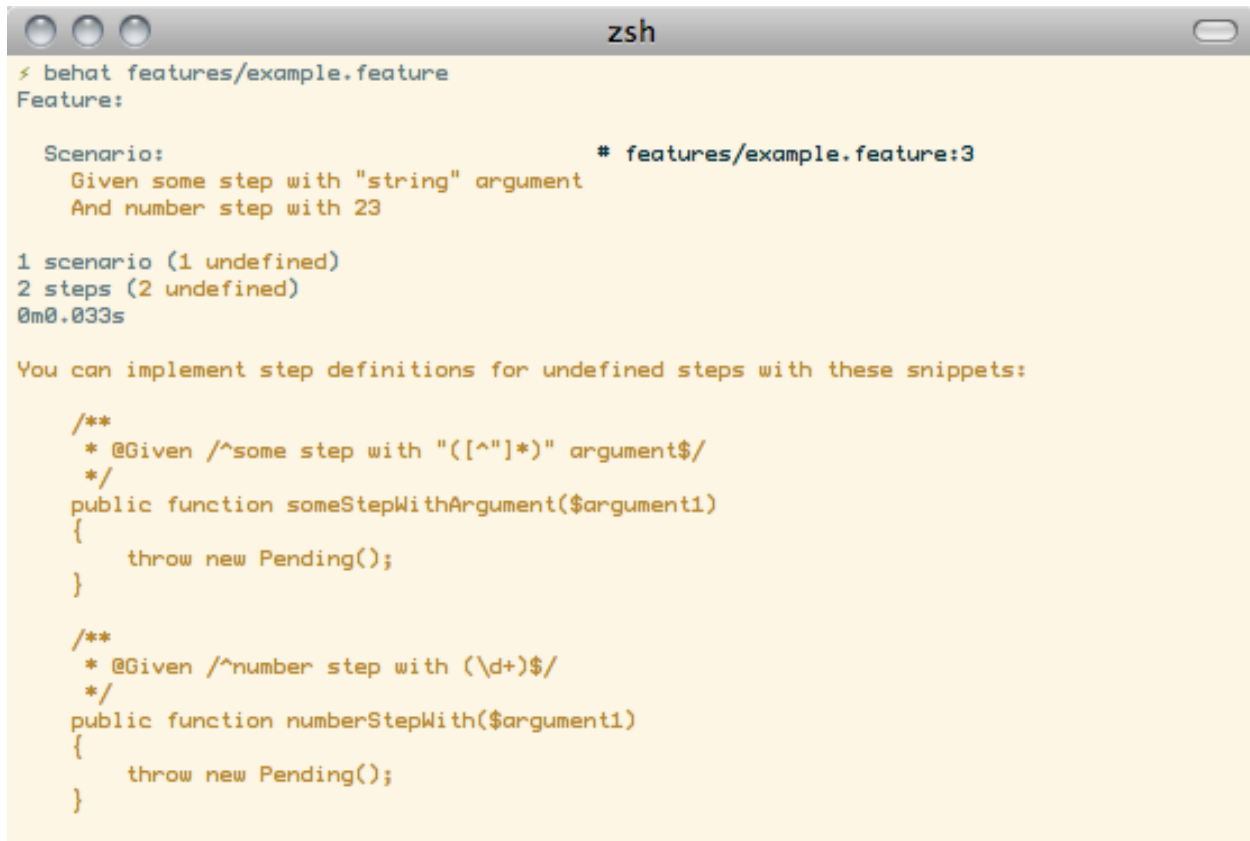
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext;

class FeatureContext extends BehatContext
{
}

```

When you run your feature, you'll get 2 undefined steps that are marked yellow:



```
zsh
$ behat features/example.feature
Feature:

    Scenario:                                     # features/example.feature:3
        Given some step with "string" argument
        And number step with 23

1 scenario (1 undefined)
2 steps (2 undefined)
0m0.033s

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

/**
 * @Given /^some step with "([^"]*)" argument$/
 */
public function someStepWithArgument($argument1)
{
    throw new Pending();
}

/**
 * @Given /^number step with (\d+)$/
 */
public function numberStepWith($argument1)
{
    throw new Pending();
}
```

---

**Note:** Undefined steps are always marked as **yellow** if colors are supported by your console.

---

**Note:** All steps following an undefined step are not executed, as the behavior following it is unpredictable. These steps are marked as **skipped**.

---

**Tip:** If you use the `--strict` option with Behat, undefined steps will cause Behat to exit with 1 code.

---

## Pending Steps

When a definition method throws a `Behat\Behat\Exception\PendingException` exception, the step is marked as **pending**, reminding you that you have work to do.

Let's pretend your `FeatureContext` looks like this:

```
<?php
```

```
use Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;

class FeatureContext extends BehatContext
{
    /** @Given /^some step with "([^"]*)" argument$/ */
    public function someStepWithArgument($argument1)
    {
        throw new PendingException('Do some string work');
    }
}
```

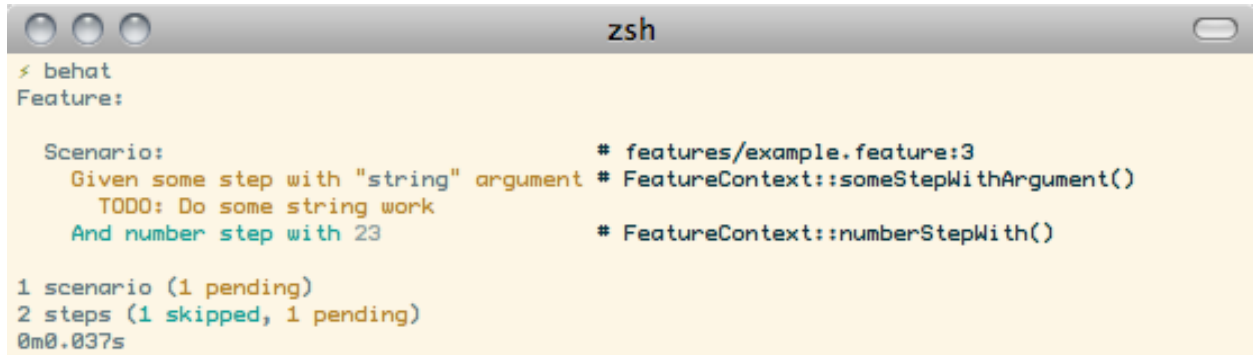


```

/** @Given /^number step with (\d+)$/ */
public function numberStepWith($argument1)
{
    throw new PendingException('Do some number work');
}

```

When you run your feature, you'll get 1 pending step that is marked yellow:



```

zsh
$ behat
Feature:

Scenario:                                     # features/example.feature:3
  Given some step with "string" argument # FeatureContext::someStepWithArgument()
  TODO: Do some string work
  And number step with 23                  # FeatureContext::numberStepWith()

1 scenario (1 pending)
2 steps (1 skipped, 1 pending)
0m0.037s

```

**Note:** Pending steps are always marked as **yellow** if colors are supported by your console, because they are logically similar to **undefined** steps.

**Note:** All steps following a pending step are not executed, as the behavior following it is unpredictable. These steps are marked as **skipped**.

**Tip:** If you use `--strict` option with Behat, pending steps will cause Behat to exit with 1 code.

## Failed Steps

When a definition method throws a generic `Exception` (not a `PendingException`) during execution, the step is marked as **failed**. Again, what you return from a definition does not affect the passing or failing of the step. Returning `null` or `false` will not cause a step definition to fail.

Let's pretend that your `FeatureContext` has the following code:

```

<?php

use Behat\Behat\Context\BehatContext;

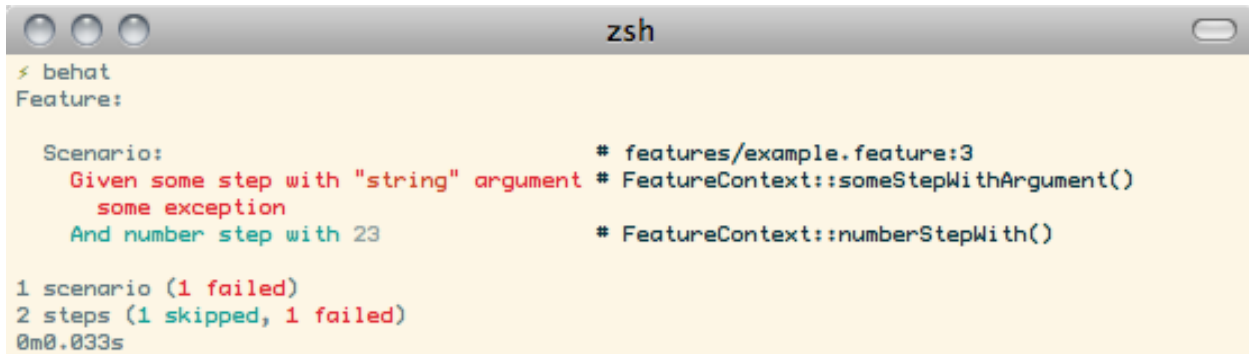
class FeatureContext extends BehatContext
{
    /** @Given /^some step with "([^"]*)" argument$/ */
    public function someStepWithArgument($argument1)
    {
        throw new Exception('some exception');
    }

    /** @Given /^number step with (\d+)$/ */
    public function numberStepWith($argument1)
    {
    }
}

```

```
}
```

When you run your feature, you'll get 1 failing step that is marked red:



```
zsh
$ behat
Feature:

  Scenario:                                     # features/example.feature:3
    Given some step with "string" argument # FeatureContext::someStepWithArgument()
      some exception
    And number step with 23                  # FeatureContext::numberStepWith()

1 scenario (1 failed)
2 steps (1 skipped, 1 failed)
0m0.033s
```

---

**Note:** Failed steps are always marked as **red** if colors are supported by your console.

---

**Note:** All steps following a failed step are not executed, as the behavior following it is unpredictable. These steps are marked as **skipped**.

---

**Tip:** If Behat finds a failed step during suite execution, it will exit with 1 code.

---

**Tip:** Behat does not come with its own assertion library, but you can use any proper assertion tool out there, as long as its failed assertions throw an exceptions. For example, if you're familiar with PHPUnit, you can use its assertions in Behat:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext;
use Behat\Gherkin\Node\PyStringNode;

require_once 'PHPUnit/Autoload.php';
require_once 'PHPUnit/Framework/Assert/Functions.php';

class FeatureContext extends BehatContext
{
    /**
     * @Then /^I should get:$/
     */
    public function iShouldGet(PyStringNode $string)
    {
        assertEquals($string->getRaw(), $this->output);
    }
}
```

---

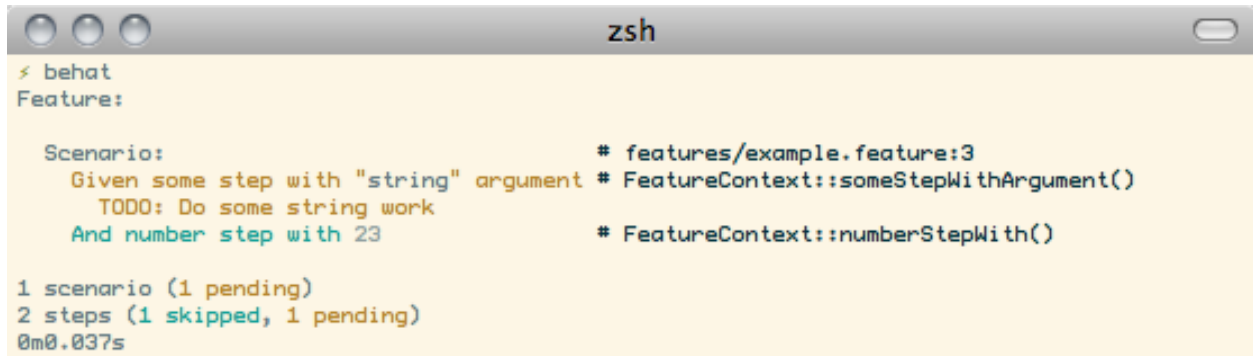
**Tip:** You can get exception stack trace with `-v` option provided to Behat:

```
$ behat features/example.feature -v
```

---

## Skipped Steps

Steps that follow **undefined**, **pending** or **failed** steps are never executed, even if there is a matching definition. These steps are marked **skipped**:



```

zsh
$ behat
Feature:

  Scenario:                                     # features/example.feature:3
    Given some step with "string" argument # FeatureContext::someStepWithArgument()
      TODO: Do some string work
    And number step with 23                  # FeatureContext::numberStepWith()

1 scenario (1 pending)
2 steps (1 skipped, 1 pending)
0m0.037s

```

**Note:** Skipped steps are always marked as **cyan** if colors are supported by your console.

## Ambiguous Steps

When Behat finds two or more definitions that match a single step, this step is marked as **ambiguous**.

Consider your FeatureContext has following code:

```

<?php

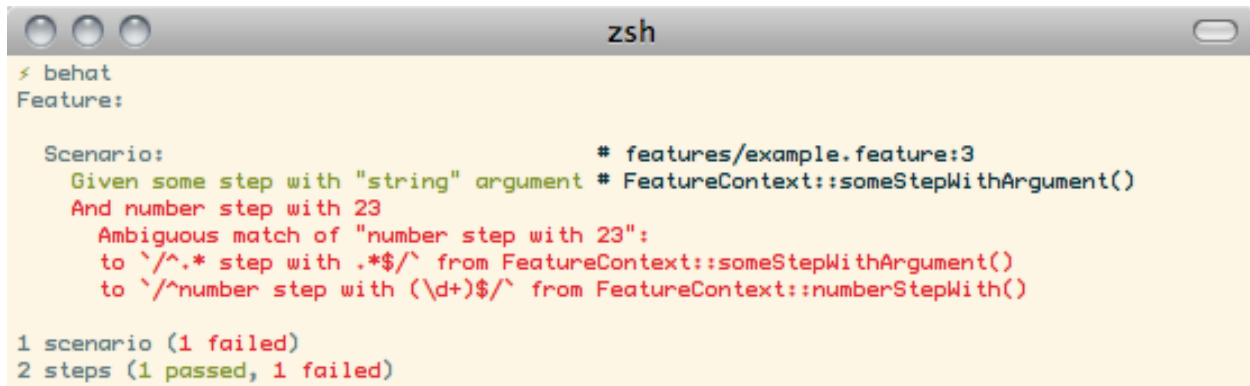
use Behat\Behat\Context\BehatContext;

class FeatureContext extends BehatContext
{
    /** @Given /^.* step with .*$/ */
    public function someStepWithArgument()
    {
    }

    /** @Given /^number step with (\d+)$/ */
    public function numberStepWith($argument1)
    {
    }
}

```

Running your feature with this context will result in:



```

zsh
$ behat
Feature:

Scenario:                                     # features/example.feature:3
  Given some step with "string" argument # FeatureContext::someStepWithArgument()
  And number step with 23
    Ambiguous match of "number step with 23":
    to `/. * step with .*$/` from FeatureContext::someStepWithArgument()
    to `/^number step with (\d+)$/` from FeatureContext::numberStepWith()

1 scenario (1 failed)
2 steps (1 passed, 1 failed)

```

Behat will not make a decision about which definition to execute. That's your job! But as you can see, Behat will provide useful information to help you eliminate such problems.

## Redundant Step Definitions

Behat will not let you define a step expression's corresponding regular expression more than once. For example, look at the two @Given regular expressions defined in this feature context:

```
<?php
```

```
use Behat\Behat\Context\BehatContext;
```

```
class FeatureContext extends BehatContext
```

```
{
```

```
    /** @Given /^number step with (\d+)$/ */
```

```
    public function workWithNumber($number1)
```

```
    {
```

```
    }
```

```
    /** @Given /^number step with (\d+)$/ */
```

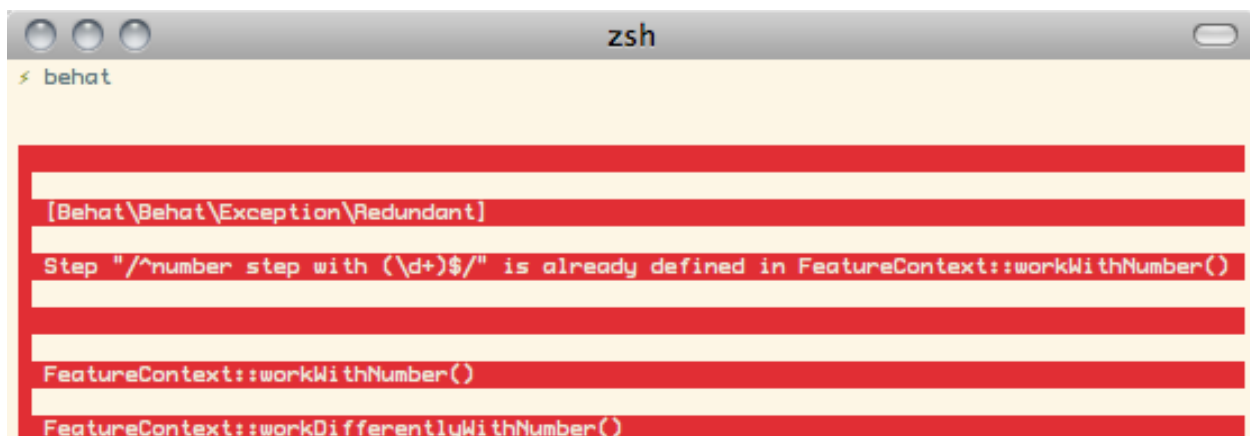
```
    public function workDifferentlyWithNumber($number1)
```

```
    {
```

```
    }
```

```
}
```

Executing Behat with this feature context will result in a Redundant exception being thrown:



```

zsh
$ behat

[Behat\Behat\Exception\Redundant]
Step "/^number step with (\d+)$/" is already defined in FeatureContext::workWithNumber()

FeatureContext::workWithNumber()
FeatureContext::workDifferentlyWithNumber()

```

## 2.2.5 Step Argument Transformations

Step argument transformations allow you to abstract common operations performed on step definition arguments into reusable methods. In addition, these methods can be used to transform a normal string argument that was going to be used as an argument to a step definition method, into a more specific data type or object.

Each transformation method must return a new value. This value then replaces the original string value that was going to be used as an argument to a step definition method.

Transformation methods are defined using the same annotation style as step definition methods, but instead use the `@Transform` keyword, followed by a matching regular expression.

As a basic example, you can automatically cast all numeric arguments to integers with the following context class code:

```
<?php

use Behat\Behat\Context\BehatContext;

class FeatureContext extends BehatContext
{
    /**
     * @Transform /^(\d+)$/
     */
    public function castStringToNumber($string)
    {
        return intval($string);
    }

    /**
     * @Then /^a user '([^']+)', should have (\d+) followers$/
     */
    public function assertUserHasFollowers($name, $count)
    {
        if ('integer' !== gettype($count)) {
            throw new Exception('Integer expected');
        }
    }
}
```

Let's go a step further and create a transformation method that takes an incoming string argument and returns a specific object. In the following example, our transformation method will be passed a username, and the method will create and return a new `User` object:

```
<?php

use Behat\Behat\Context\BehatContext;

class FeatureContext extends BehatContext
{
    /**
     * @Transform /^user (.*)$/
     */
    public function castUsernameToUser($username)
    {
        return new User($username);
    }

    /**
```

```
* @Then /^a '(user [^']+)', should have (\d+) followers$/
*/
public function assertUserHasFollowers(User $name, $count)
{
    if ('integer' !== gettype($count)) {
        throw new Exception('Integer expected');
    }
}
}
```

## Transforming Tables

Let's pretend we have written the following feature:

```
# features/table.feature
Feature: Users
```

```
Scenario: Creating Users
```

```
Given the following users:
```

name	followers
everzet	147
avalanche123	142
kriswallsmith	274
fabpot	962

And our FeatureContext class looks like this:

```
<?php
```

```
use Behat\Behat\Context\BehatContext;
use Behat\Gherkin\Node\TableNode;
```

```
class FeatureContext extends BehatContext
{
    /**
     * @Given /^the following users$/
     */
    public function pushUsers(TableNode $usersTable)
    {
        $users = array();
        foreach ($usersTable->getHash() as $userHash) {
            $user = new User();
            $user->setUsername($userHash['name']);
            $user->setFollowersCount($userHash['followers']);
            $users[] = $user;
        }

        // do something with $users
    }
}
```

A table like this may be needed in a step testing the creation of the `User` objects themselves, and later used again to validate other parts of our codebase that depend on multiple `User` objects that already exist. In both cases, our transformation method can take our table of usernames and follower counts and build dummy `User` objects. By using a transformation method we have eliminated the need to duplicate the code that creates our `User` objects, and can instead rely on the transformation method each time this functionality is needed.

Transformations can also be used with tables. A table transformation is matched via a comma-delimited list of the column headers prefixed with `table::`:

```
<?php

use Behat\Behat\Context\BehatContext;
use Behat\Gherkin\Node\TableNode;

class FeatureContext extends BehatContext
{
    /**
     * @Transform /^table:name, followers$/
     */
    public function castUsersTable(TableNode $usersTable)
    {
        $users = array();
        foreach ($usersTable->getHash() as $userHash) {
            $user = new User();
            $user->setUsername($userHash['name']);
            $user->setFollowersCount($userHash['followers']);
            $users[] = $user;
        }

        return $users;
    }

    /**
     * @Given /^the following users$/
     */
    public function pushUsers(array $users)
    {
        // do something with $users
    }

    /**
     * @Then /^I expect the following users$/
     */
    public function assertUsers(array $users)
    {
        // do something with $users
    }
}
```

---

**Note:** Transformations are powerful and it is important to take care how you implement them. A mistake can often introduce strange and unexpected behavior.

---

## 2.2.6 Step Execution Chaining

Sometimes it might be useful to pass execution flow from one step to another. For example, if during step definition execution you found that it might be better to call another step to keep from duplicating code, you can just return a step imitator object (substep) from a definition method:

```
<?php

use Behat\Behat\Context\BehatContext,
    Behat\Behat\Context\Step\Then;
```

```
use Behat\Gherkin\Node\TableNode;

class FeatureContext extends BehatContext
{
    /**
     * @Then /^(?:|I )should be on "(?P<page>[^\"]+)"$/
     */
    public function assertPageAddress($page)
    {
        // check, that $page is equal to current page
    }

    /**
     * @Then /^the url should match "(?P<pattern>[^\"]+)"$/
     */
    public function assertUrlRegExp($pattern)
    {
        if (!preg_match('/^\./.*\./', $pattern)) {
            return new Then("I should be on \"$pattern\"");
        }

        // do regex assertion
    }
}
```

Notice that when we do not provide a regular expression to Then the url should match "... " in the step definition `assertUrlRegExp()`, it returns a new `Behat\Behat\Context\Step\Then` instance. When a step definition returns such object, it finds and executes the step definition that matches the step text provided as that object's argument.

---

**Tip:** There are three predefined substep classes you can to use:

1. `Behat\Behat\Context\Step\Given`
2. `Behat\Behat\Context\Step\When`
3. `Behat\Behat\Context\Step\Then`

These are the same steps used to annotate step definitions.

---

You can also return steps with multiline arguments. You can even pass in a table as an argument:

```
/**
 * @Given /^I have the initial table$/
 */
public function table()
{
    $table = new Behat\Gherkin\Node\TableNode(<<<TABLE
        | username | password |
        | everzet   | 123456   |
TABLE
    );

    return new Given('I have users:', $table);
}
```

---

**Note:** Steps executed as a chain will throw an exception for all result types except for **Successful**. This means you'll never get snippets out of steps, called **only** through execution chain!

---



As of 2.0.4, if you want to pass more than one step in an execution chain, just return an array of substep instances:

```
/**
 * @Given /I entered "([^"]*)" and expect "([^"]*)" /
 */
public function complexStep($number, $result)
{
    return array(
        new Step\Given("I have entered \"$number\""),
        new Step\When("I press +"),
        new Step\Then("I should see \"$result\" on the screen")
    );
}
```

## 2.3 Hooking into the Test Process - Hooks

You've learned *how to write step definitions* and that with *Gherkin* you can move common steps into the background block, making your features DRY. But what if that's not enough? What if you want to execute some code before the whole test suite or after a specific scenario? Hooks to the rescue:

```
# features/bootstrap/FeatureContext.php
<?php

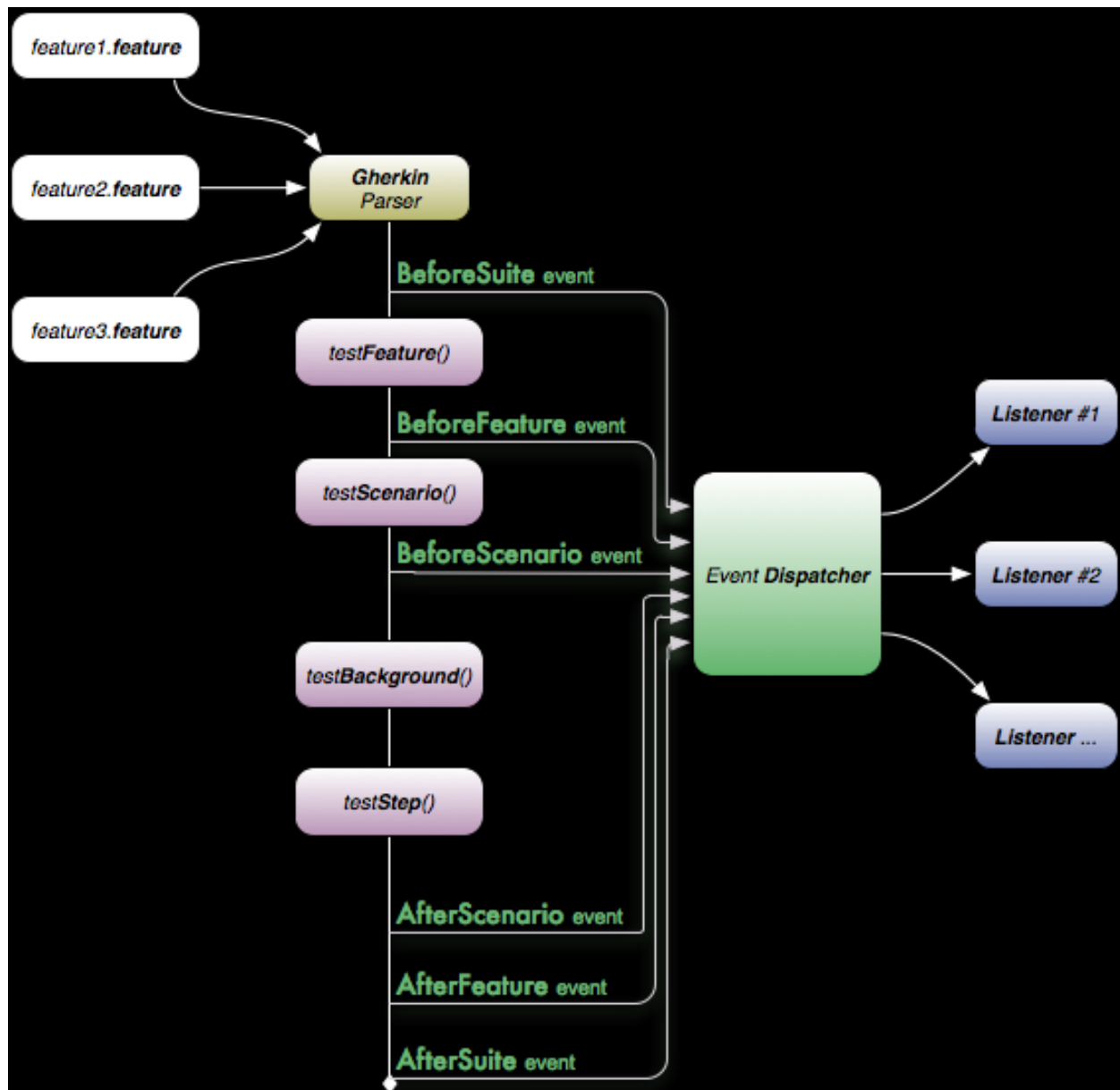
use Behat\Behat\Context\BehatContext,
    Behat\Behat\Event\SuiteEvent,
    Behat\Behat\Event\ScenarioEvent;

class FeatureContext extends BehatContext
{
    /**
     * @BeforeSuite
     */
    public static function prepare(SuiteEvent $event)
    {
        // prepare system for test suite
        // before it runs
    }

    /**
     * @AfterScenario @database
     */
    public function cleanDB(ScenarioEvent $event)
    {
        // clean database after scenarios,
        // tagged with @database
    }
}
```

### 2.3.1 Behat Event System

Before understanding what hooks really are and how to use them, you should first understand how Behat actually works (on the highest possible level). Consider this diagram:



Let's explain it a little bit:

1. First, Behat reads all features files that you provide to it.
2. Second, Behat passes the loaded feature files to the Gherkin parser, which builds an abstract syntax tree out of every one of them.
3. Next, Behat passes every parsed feature tree into the feature tester.
4. The feature tester retrieves all scenarios in the passed feature and passes them further along to the scenario tester.
5. The scenario tester initializes a context object out of your `FeatureContext` class and passes it, with all scenario step nodes and background step nodes (if the feature has them), further to the step tester.
6. The step tester searches for matching step definitions in the provided context object and, if it finds one, executes it.

That's basically it! Behat exits when it finishes executing the last step in the last scenario of the last parsed feature.

Here's a good question for you to consider: how does Behat print suite execution information into the console and collect statistics to be printed after? The answer is hidden in the [Observer Design Pattern](#).

On every point of execution, testers initialize special objects called *events* (in green text in the diagram) and send them to a special `EventDispatcher` object (green square in the diagram). Other objects, called *listeners* (blue squares in the diagram) register on `EventDispatcher` to be able to receive those events. `EventDispatcher` automatically dispatches received events to registered listeners that are capable of handling a particular event type.

That's how statistic collectors, formatters and hooks work. They just provide listeners to some type of Behat lifetime events.

## 2.3.2 Hooks

Behat hooks are a simple way to execute code when core Behat events occur. Behat provides 8 event types for you to hook into:

1. The `BeforeSuite` event happens before any feature in the suite runs. For example, you could use this to setup the project you are testing. This event comes with an instance of the `Behat\Behat\Event\SuiteEvent` class.
2. The `AfterSuite` event happens after all features in the suite have run. This event is used internally for statistics printing. This event comes with an instance of the `Behat\Behat\Event\SuiteEvent` class.
3. The `BeforeFeature` event occurs before a feature runs. This event comes with an instance of the `Behat\Behat\Event\FeatureEvent` class.
4. The `AfterFeature` event occurs after Behat finishes executing a feature. This event comes with an instance of the `Behat\Behat\Event\FeatureEvent` class.
5. The `BeforeScenario` event occurs before a specific scenario will run. This event comes with an instance of the `Behat\Behat\Event\ScenarioEvent` class.
6. The `AfterScenario` event occurs after Behat finishes executing a scenario. This event comes with an instance of the `Behat\Behat\Event\ScenarioEvent` class.
7. The `BeforeStep` event occurs before a specific step runs. This event comes with an instance of the `Behat\Behat\Event\StepEvent` class.
8. The `AfterStep` event occurs after Behat finishes executing a step. This event comes with an instance of the `Behat\Behat\Event\StepEvent` class.

You can hook into any of these events by using annotations on methods in your `FeatureContext` class:

```
/**
 * @BeforeSuite
 */
public static function prepare(SuiteEvent $event)
{
    // prepare system for test suite
    // before it runs
}
```

We use annotations as we did before with definitions. Simply use the annotation of the name of the event you want to hook into (e.g. `@BeforeSuite`).

## 2.3.3 Suite Hooks

Suite hooks are triggered before or after actual scenarios, so `FeatureContext` is used. So suite hooks should be defined as `public static` methods in your `FeatureContext` class:

```
/** @BeforeSuite */
public static function setup(SuiteEvent $event)
{
}

/** @AfterSuite */
public static function teardown(SuiteEvent $event)
{
}
```

There are two suite hook types available:

- @BeforeSuite - executed before any feature runs.
- @AfterSuite - executed after all features have run.

Both hooks receive `Behat\Behat\Event\SuiteEvent` as their argument. This object has some useful methods for you to consider:

- `getContextParameters()` - returns an array of parameters for your context instance.
- `getLogger()` - returns `Behat\Behat\DataCollector\LoggerDataCollector` instance, which holds all suite run statistics.
- `isCompleted()` - returns true when the whole suite is successfully executed, or false when the suite is not executed (@BeforeSuite or @AfterSuite after SIGINT).

## 2.3.4 Feature Hooks

Feature hooks are triggered before or after each feature runs. Like [Suite Hooks](#), a `FeatureContext` instance is not created. Feature hooks should also be defined as `public static` methods:

```
/** @BeforeFeature */
public static function setupFeature(FeatureEvent $event)
{
}

/** @AfterFeature */
public static function teardownFeature(FeatureEvent $event)
{
}
```

There are two feature hook types available:

- @BeforeFeature - gets executed before every feature in the suite.
- @AfterFeature - gets executed after every feature in the suite.

Both hooks receive `Behat\Behat\Event\FeatureEvent` as their argument. This object has useful methods for you:

- `getContextParameters()` - returns an array of parameters for your context instance.
- `getFeature()` - returns a `Behat\Gherkin\Node\FeatureNode` instance, which is an abstract syntax tree representing the whole feature.
- `getResult()` - returns the resulting (highest) feature run code: 4 when the feature has failed steps, 3 when the feature has undefined steps, 2 when the feature has pending steps and 0 when all steps are passing.

### 2.3.5 Scenario Hooks

Scenario hooks are triggered before or after each scenario runs. These hooks are executed inside an initialized `FeatureContext` instance, so they are just plain `FeatureContext` instance methods:

```
/** @BeforeScenario */
public function before($event)
{
}

/** @AfterScenario */
public function after($event)
{
}
```

There are two scenario hook types available:

- `@BeforeScenario` - executed before every scenario in each feature.
- `@AfterScenario` - executed after every scenario in each feature.

Now, the interesting part:

The `@BeforeScenario` hook executes not only before each scenario in each feature, but before **each example row** in the scenario outline. Yes, each scenario outline example row works almost the same as a usual scenario, except that it sends a different event: `Behat\Behat\Event\OutlineExampleEvent`.

`@AfterScenario` functions exactly the same way, except after each scenario in each feature.

So, the `@BeforeScenario` or `@AfterScenario` hook will receive either a `Behat\Behat\Event\ScenarioEvent` or `Behat\Behat\Event\OutlineExampleEvent` instance, depending on the situation. It's your job to differentiate between them if needed.

`Behat\Behat\Event\ScenarioEvent` has the following methods:

- `getScenario()` - returns a `Behat\Gherkin\Node\ScenarioNode` instance, which is an abstract syntax tree node representing a specific scenario.
- `getContext()` - returns a `FeatureContext` instance. But take note! Because your hook method is already defined inside `FeatureContext`, the `FeatureContext` instance passed with the event is **the same object** accessible from within the method itself by using `$this->`. The instance passed with the event isn't very useful in this case.
- `getResult()` - returns the resulting (highest) step run code. 4 when the scenario has failed steps, 3 when the scenario has undefined steps, 2 when the scenario has pending steps and 0 when all steps are passing.
- `isSkipped()` - returns `true` if the scenario has skipped steps (steps that follow **pending, undefined** or **failed** steps).

`Behat\Behat\Event\OutlineExampleEvent` has the following methods:

- `getOutline()` - returns a `Behat\Gherkin\Node\OutlineNode` instance, which is an abstract syntax tree node representing a specific scenario outline.
- `getIteration()` - returns an integer representing the example row number that sent this event.
- `getContext()` - returns a `FeatureContext` instance. But take note! Because your hook method is already defined inside `FeatureContext`, the `FeatureContext` instance passed with the event is **the same object** accessible from within the method itself by using `$this->`. The instance passed with the event isn't very useful in this case.

- `getResult()` - returns the resulting (highest) step run code. This returned value is one of `StepEvent` constants: 4 when an examples row has failed steps, 3 when a row has undefined steps, 2 when a row has pending steps and 0 when all steps are passing.
- `isSkipped()` - returns `true` if the scenario has skipped steps (steps that follow **pending**, **undefined** or **failed** steps).

## 2.3.6 Step Hooks

Step hooks are triggered before or after each step runs. These hooks are executed inside an initialized `FeatureContext` instance, so they are just plain `FeatureContext` instance methods:

```
/** @BeforeStep */
public function beforeStep(StepEvent $event)
{
}

/** @AfterStep */
public function after(StepEvent $event)
{
}
```

There are two step hook types available:

- `@BeforeStep` - executed before every step in each scenario.
- `@AfterStep` - executed after every step in each scenario.

Both hooks receive `Behat\Behat\Event\StepEvent` as their argument. This object has the following methods:

- `getStep()` - returns a `Behat\Gherkin\Node\StepNode` instance, which is an abstract syntax tree node representing a scenario step.
- `getContext()` - returns a `FeatureContext` instance. But take note! Because your hook method is already defined inside `FeatureContext`, the `FeatureContext` instance passed with the event is **the same object** accessible from within the method itself by using `$this->`. The instance passed with the event isn't very useful in this case.
- `getResult()` - returns the resulting step run code. 4 when the step has failed, 3 when the step is undefined, 2 when the step is pending, 1 when the step has been skipped and 0 when the step is passing.
- `hasDefinition()` - returns `true` if a definition for the current step is found.
- `getDefinition()` - returns `Behat\Behat\Definition\DefinitionInterface` implementation, which represents the matched step definition for this step.
- `hasException()` - returns `true` if the step threw an exception during its execution.
- `getException()` - returns an exception instance thrown from within the step, if one was thrown.
- `hasSnippet()` - returns `true` if the step is undefined.
- `getSnippet()` - returns the step snippet if the step is undefined.

## 2.3.7 Tagged Hooks

Sometimes you may want a certain hook to run only for certain scenarios, features or steps. This can be achieved by associating a `@BeforeFeature`, `@AfterFeature`, `@BeforeScenario`, `@AfterScenario`, `@BeforeStep` or `@AfterStep` hook with one or more tags. You can also use `OR (|)` and `AND (&)` tags:

```
/**
 * @BeforeScenario @database,@orm
 */
public function cleanDatabase()
{
    // clean database before
    // @database OR @orm scenarios
}
```

Use the && tag to execute a hook only when it has *all* provided tags:

```
/**
 * @BeforeScenario @database&&@fixtures
 */
public function cleanDatabaseFixtures()
{
    // clean database fixtures
    // before @database @fixtures
    // scenarios
}
```

## 2.4 Testing Features - FeatureContext Class

We've already used this strange FeatureContext class as a home for our *step definitions* and *hooks*, but we haven't done much to explain what it actually is.

The Context class is a simple POPO (Plain Old PHP Object) used by Behat to represent testing part of your features suite. If \*.feature files are all about describing *how* your application behaves, then the context class is all about how to test your application, and that it actually behaves as expected. That's it!

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext;

require_once 'PHPUnit/Autoload.php';
require_once 'PHPUnit/Framework/Assert/Functions.php';

class FeatureContext extends BehatContext
{
    public function __construct(array $parameters)
    {
        $this->useContext('subcontext_alias', new AnotherContext());
    }

    /** @BeforeFeature */
    public static function prepareForTheFeature()
    {} // clean database or do other preparation stuff

    /** @Given /^we have some context$/ */
    public function prepareContext()
    {} // do something

    /** @When /^event occurs$/ */
    public function doSomeAction()
    {} // do something
```

```
/** @Then /^something should be done$/ */
public function checkOutcomes()
{ } // do something
}
```

## 2.4.1 Context Class Requirements

In order to be used by Behat, your context class should follow 3 simple rules:

1. Context class should implement `Behat\Behat\Context\ContextInterface` or extend base class `Behat\Behat\Context\BehatContext` (as in the previous example).
2. Context class should be called `FeatureContext`. It's a simple convention inside the Behat infrastructure.
3. Context class should be findable and loadable by Behat. That means you should somehow tell Behat about your class file. The easiest way to do this is to put your class file inside the `features/bootstrap/` directory. All `*.php` files in this directory are autoloaded by Behat before any feature is run.

---

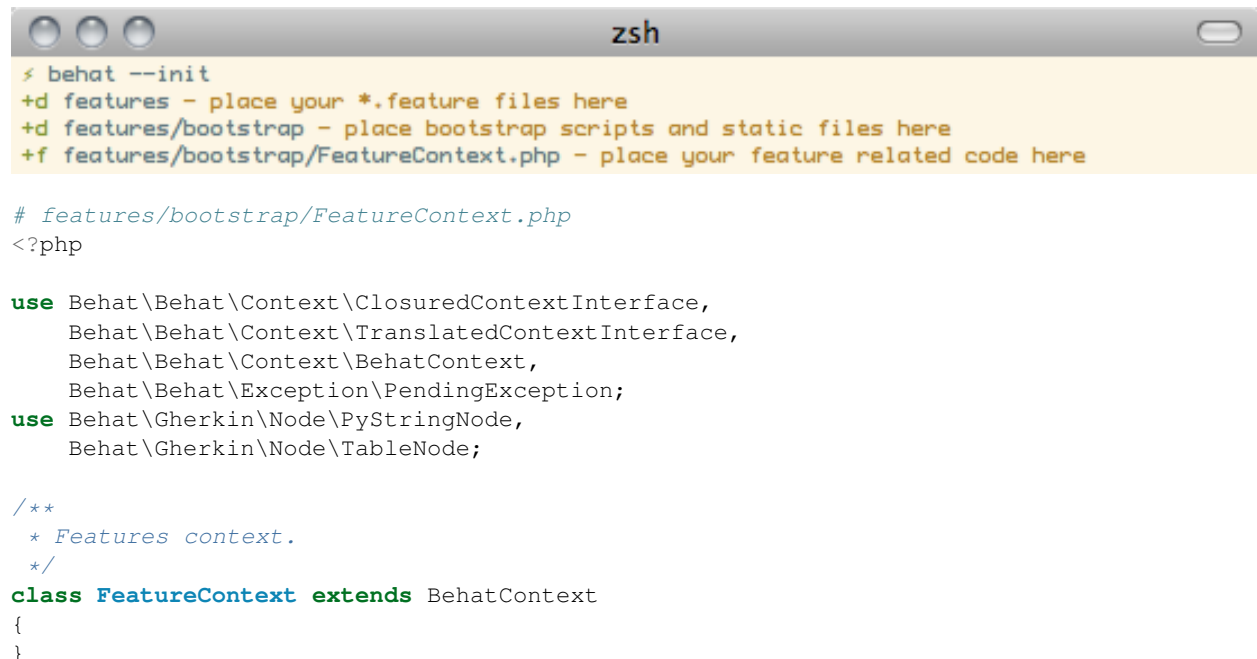
**Note:** By convention, the context class should be called `FeatureContext`, but this could be easily changed through the [cli](#) configuration.

---

The easiest way to start using Behat in your project is to call `behat` with the `--init` option inside your project directory:

```
$ behat --init
```

Behat will create a few directories and a skeleton `FeatureContext` class inside your project:



```
zsh
$ behat --init
+ d features - place your *.feature files here
+ d features/bootstrap - place bootstrap scripts and static files here
+ f features/bootstrap/FeatureContext.php - place your feature related code here

# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\TranslatedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

/**
 * Features context.
 */
class FeatureContext extends BehatContext
{
}
```

## 2.4.2 Contexts Lifetime

Your context class is initialized before each scenario runs, and every scenario has its very own context instance. This means 2 things:



1. Every scenario is isolated from each other scenario's context. You can do almost anything inside your scenario context instance without the fear of affecting other scenarios - every scenario gets its own context instance.
2. Every step in a single scenario is executed inside a common context instance. This means you can set `private` instance variables inside your `@Given` steps and you'll be able to read their new values inside your `@When` and `@Then` steps.

### 2.4.3 Using Subcontexts

At some point, it could become very hard to maintain all your *step definitions* and *hooks* inside a single class. You could use class inheritance and split definitions into multiple classes, but doing so could cause your code to become more difficult to follow and use.

In light of these issues, Behat provides a more flexible way to help make your code more reusable: using one or more contexts inside your main context. A context used from within another context is called a *subcontext*:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext;

class FeatureContext extends BehatContext
{
    public function __construct(array $parameters)
    {
        $this->useContext('subcontext_alias', new SubContext());
    }
}
```

---

**Note:** PHP does not yet support horizontal reusability in its core feature set. While this functionality, called *traits*, is on the roadmap for PHP 5.4, Behat provides subcontexts as a stop-gap solution to achieve horizontal reusability until this functionality is available in a stable PHP release.

---

`Behat\Behat\Context\BehatContext` provides a special `useContext()` instance method allowing you to connect one or more subcontext instances to your main `FeatureContext` class.

The first argument to the `useContext()` method is always a subcontext alias (`subcontext_alias`), allowing you to later access any subcontext from another subcontext.

`SubContext` instances should follow the same [Context Class Requirements](#) as your main `FeatureContext`:

```
# features/bootstrap/SubContext.php
<?php

use Behat\Behat\Context\BehatContext;

class SubContext extends BehatContext
{
    public function __construct(array $parameters)
    {
        // do subcontext initialization
    }
}
```

All *step definitions* and *hooks* defined in a subcontext are parsed by Behat and available right away to use in your features.

If you need to inject parameters or make other changes to your subcontext object, do so before passing it into `useContext()`:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\BehatContext;

class FeatureContext extends BehatContext
{
    public function __construct(array $parameters)
    {
        $this->useContext('subcontext_alias', new SubContext(array(
            /* custom params */
        )));
    }
}
```

## Communications Between Contexts

Sometimes you might need to call a specific context method or attribute from within another context. `BehatContext` has two methods to accomplish this:

1. `getMainContext()` - returns the main context instance in which all other contexts are used.
2. `getSubcontext($alias)` - returns a subcontext given its alias, which was defined when it was passed to `useContext()`.

Keeping this in mind, you can always call any context method using the following statement:

```
$this->getMainContext()->getSubcontext('subcontext_alias')->some_method();
```

## 2.4.4 Creating Your Very Own Context Class

The easiest way to start with Behat is to just extend the base class `Behat\Behat\Context\BehatContext`. But what if you don't want to inherit from another class? Then you should create your own context class.

To use your custom class as a Behat context, it must implement a simple interface:

```
<?php

namespace Behat\Behat\Context;

interface ContextInterface
{
    function getSubcontexts();
    function getSubcontextByClassName($className);
}
```

This interface actually only has 2 methods:

- `getSubcontexts()` - should return an array of subcontext instances (if it even has any).
- `getSubcontextByClassName()` - should return a subcontext instance given its class name. This method is used to ensure your subcontext definitions will always be called inside the proper context instance.

Your custom `FeatureContext` class could look like this:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\ContextInterface;

class FeatureContext implements ContextInterface
{
    private $subcontext;

    public function __construct()
    {
        $this->subcontext = new SubContext();
    }

    public function getSubcontexts()
    {
        return array($this->subcontext);
    }

    public function getSubcontextByClassName($className)
    {
        if ('SubContext' === $className) {
            return $this->subcontext;
        }
    }
}
```

## 2.5 Closures as Definitions and Hooks

The official best way to add *step definitions* and *hooks* is to write them as class or instance methods of a *context class*. But step definitions can be written in other ways too.

Some people dislike the verbosity of object-oriented PHP, where developers need to worry about method visibility (public, private) and whether to declare a method as an instance method or as `static`. On top of that, we have used annotations to tell Behat how our methods should be used. If you would like a more concise way to declare your step definitions, you're in luck!

```
<?php

$steps->Given('/^I have ordered hot "([^"]*)"$/ ',
    function($world, $arg1) {
        throw new Behat\Behat\Exception\PendingException();
    }
);

$steps->When('/^the "([^"]*)" will be ready$/ ',
    function($world) {
        throw new Behat\Behat\Exception\PendingException();
    }
);
```

### 2.5.1 Closed Context

In order to use closures as definitions or hooks in your suite, you'll need to extend your `FeatureContext` a little bit. To be able to load your closure files, Behat will need a way to actually find them first.

To use closures, your `FeatureContext` must implement the interface `Behat\Behat\Context\ClosedContextInterface`:

```
<?php

namespace Behat\Behat\Context;

interface ClosedContextInterface extends ContextInterface
{
    function getStepDefinitionResources();
    function getHookDefinitionResources();
}
```

There are only two methods in this interface:

- `getStepDefinitionResources()` - should return an array of file paths pointing to `*.php` files to be used as **step definition** resources.
- `getHookDefinitionResources()` - should return an array of file paths pointing to `*.php` files to be used as **hook definition** resources.

The following example `FeatureContext` implements this interface:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\BehatContext;

/**
 * Features context.
 */
class FeatureContext extends BehatContext implements ClosedContextInterface
{
    public function getStepDefinitionResources()
    {
        return array(__DIR__ . '/../steps/basic_steps.php');
    }

    public function getHookDefinitionResources()
    {
        return array(__DIR__ . '/../support/hooks.php');
    }
}
```

Given this example, Behat will try to load **step definitions** from `features/steps/basic_steps.php` and **hooks** from `features/support/hooks.php`.

## 2.5.2 Step Definitions

Every `*.php` path returned by `getStepDefinitionResources()` will be loaded with the variable `$steps` already defined.

Use the provided `$steps` variable to define *step definitions*:

```
<?php

$steps->Given('/^I have ordered hot "([^"]*)"$/ ',
    function($world, $arg1) {
```

```

        throw new Behat\Behat\Exception\PendingException();
    }
};

```

In the previous example, we call a *definition generator*. This generator maps the provided closure to the given regular expression.

Just like their annotation counterparts, Behat does not distinguish between keyword methods (Given, When, Then) available via `$steps`, and uses them only to make your definition files more readable. In fact, the name of the method doesn't matter one bit!

```

<?php

$steps->SomeUnexistentKeyword('/^I have ordered hot "[^"]*"$/ ',
    function($world, $arg1) {
        throw new Behat\Behat\Exception\PendingException();
    }
);

```

The first argument to the definition generator is a regular expression, and the second argument is a closure that would be called when the regular expression matches your *Gherkin* step.

The first argument to the provided closure should **always** be an instance of `FeatureContext`. This is done for you to be able to share context information between scenario steps. Classes in PHP have `$this`, but closures have no concept of `$this` (at least until PHP 5.4):

```

<?php

$steps->Given('/^some context$/ ', function($world) {
    $world->someVar = 'someVal';
});

$steps->Then('/^outcome$/ ', function($world) {
    // $world->someVar === 'someVal'
});

```

**Note:** `$world` is always an instance of the **main** `FeatureContext` class. This means you should provide missing methods and properties for your *subcontexts*:

```

# features/bootstrap/FeatureContext.php
<?php

class FeatureContext
{
    public function __construct(array $parameters)
    {
        $this->useContext(new SubContext($this));
    }

    public function doSomething()
    {
        // ...
    }
}

# features/bootstrap/SubContext.php
<?php

class SubContext

```

```
{
    private $mainContext;

    public function __construct (FeatureContext $context)
    {
        $this->mainContext = $context;
    }

    public function doSomething()
    {
        $this->mainContext->doSomething();
    }
}
```

---

### 2.5.3 Hooks

Every \*.php path returned by `getHookDefinitionResources()` will be loaded with the variable `$hooks` already defined.

Use the `$hooks` variable to define your *hooks*:

```
<?php

$hooks->beforeFeature('', function($event) {
    // prepare feature
});

$hooks->afterFeature('', function($event) {
    // teardown feature
});
```

You have the ability to call all hook types described in the “*Hooking into the Test Process - Hooks*” chapter. The only difference is that the method names are camel-cased (e.g. `@BeforeFeature` becomes `beforeFeature()`).

The first argument to all hook generators, except `beforeSuite` and `afterSuite`, is a tag filter.

In other parts, closure hooks are the same as normal annotated hooks.

## 2.6 Command Line Tool - behat

Behat on its own is just a command line utility that executes your behavior descriptions written in *Gherkin language*. `behat` comes with bunch of useful options and commands:



```

$ behat -h
Usage:
  behat [-c|--config="..."] [-p|--profile="..."] [--init] [-f|--format="..."] [--out="..."]
  [--colors] [--no-colors] [--no-time] [--lang="..."] [--no-paths] [--no-snippets] [--no-multiline]
  [--expand] [--story-syntax] [--definitions] [--name="..."] [--tags="..."] [--rerun="..."]
  [--strict] [features]

Arguments:
  features          Feature(s) to run. Could be:
                    - a dir (features/)
                    - a feature (*.feature)
                    - a scenario at specific line (*.feature:10).

Options:
  --config (-c)     Specify external configuration file to load.
                    behat.yml or config/behat.yml will be used by default.
  --profile (-p)    Specify configuration profile to use.
                    Define profiles in config file (--config).
  --init            Create features directory structure.
  --format (-f)     How to format features. pretty is default.
                    Available formats are:
                    - pretty: Prints the feature as is.
                    - progress: Prints one character per step.
                    - html: Generates a nice looking HTML report.
                    - junit: Generates a report similar to Ant+JUnit.
  --out            Write formatter output to a file/directory
                    instead of STDOUT (output_path).
  --colors          Force Behat to use ANSI color in the output.
  --no-colors       Do not use ANSI color in the output.
  --no-time         Hide time in output.
  --lang            Print formatter output in particular language.
  --no-paths        Do not print the definition path with the steps.
  --no-snippets     Do not print snippets for undefined steps.
  --no-multiline    No multiline arguments in output.
  --expand          Expand Scenario Outline Tables in output.
  --story-syntax    Print *.feature example.
                    Use --lang to see specific language.
  --definitions     Print available step definitions.
                    Use --lang to see specific language.
  --name            Only execute the feature elements which match
                    part of the given name or regex.
  --tags            Only execute the features or scenarios with tags
                    matching tag filter expression.
  --rerun           Save list of failed scenarios into new file
                    or use existing file to run only scenarios from it.
  --strict          Fail if there are any undefined or pending steps.

```

## 2.6.1 Informational Options

To show all available commands and options, run:

```
$ behat -h
```

To view the version of the behat tool used, call:

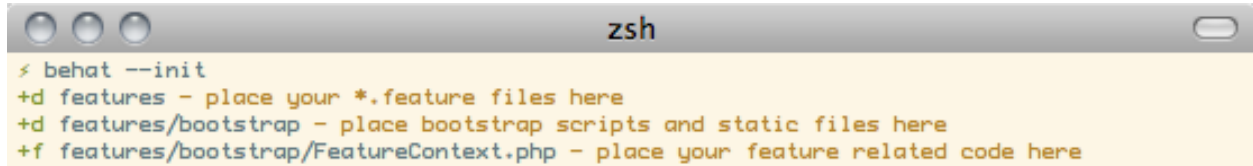
```
$ behat -V
```

### 2.6.2 Initialization Options

To initialize the feature suite structure inside your project, run:

```
$ behat --init
```

Running this command will setup a `features` directory and a skeleton `FeatureContext` class inside your project:

A terminal window titled 'zsh' showing the output of the command 'behat --init'. The output lists three directories to be created: 'features' for feature files, 'features/bootstrap' for bootstrap scripts, and 'features/bootstrap/FeatureContext.php' for feature-related code.

```
zsh
$ behat --init
+d features - place your *.feature files here
+d features/bootstrap - place bootstrap scripts and static files here
+f features/bootstrap/FeatureContext.php - place your feature related code here
```

You can use *configuration files* to configure your feature suite. By default, behat will try to load `behat.yml` and `config/behat.yml`, but if you want to name your config file differently, tell behat about it with the `--config` option:

```
$ behat --config custom-config-file.yml
```

Your *configuration files* can have multiple profiles (named configurations). You can run behat with a specific profile by calling it with the `--profile` option:

```
$ behat --config behat.yml --profile ios
```

---

**Note:** The default profile is always named `default`.

---

### 2.6.3 Format Options

Behat supports different ways of printing output information. Output printers in behat are called *formats* or *formatters*. You can tell behat to run with a specific formatter by providing the `--format` option:

```
$ behat --format progress
```

---

**Note:** The default formatter is `pretty`.

---

behat supports 6 formatters out of the box:

- `pretty` - prints the feature as is:



```

$ behat --format pretty
Feature: Some feature
  In order to ...
  As a ...
  I need ...

  Scenario: # features/example.feature:6
    Given some context # FeatureContext::someContext()
    When I do something # FeatureContext::iDoSomething()
    Some exception
    Then I should see # FeatureContext::iShouldSee()

1 scenario (1 failed)
3 steps (1 passed, 1 skipped, 1 failed)
0m0.035s

```

- progress - prints one character per step:

```

$ behat --format progress
.F-

(::$) failed steps (:::)

01. Some exception
  In step `When I do something`. # FeatureContext::iDoSomething()
  From scenario ***,           # features/example.feature:6

1 scenario (1 failed)
3 steps (1 passed, 1 skipped, 1 failed)
0m0.032s

```

- html - almost the same as pretty, but prints HTML output.
- junit - generates a report similar to Ant+JUnit.
- failed - prints paths to failed scenarios.
- snippets - prints only snippets for undefined steps.

If you don't want to print output to the console, you can tell behat to print output to a file instead of STDOUT with the `--out` option:

```
$ behat --format html --out report.html
```

**Note:** Some formatters, like `junit`, always require the `--out` option to be specified. The `junit` formatter generates `*.xml` files for every feature, so it needs a destination directory to put these XML files into.

Also, you can specify multiple formats to be used by Behat with comma (,):

```
$ behat -f pretty,progress
```

In this case, default output will be used as output for both formatters. But if you want them to use different ones - specify them with `--out`:

```
$ behat -f pretty,progress,junit --out ~/pretty.out,,xml
```

In this case, output of pretty formatter will be written to `~/pretty.out` file, output of junit formatter will be written to `xml` folder and progress formatter will just print to console. Empty out option (as in case with progress in example) tells Behat to use stdout. So:

```
$ behat -f pretty,progress,junit --out ,progress.out,xml
```

Will print pretty output instead, but will write progress output to `progress.out` file.

Behat tries hard to identify if your terminal supports colors or not, but sometimes it still fails. In such cases, you can force behat to use colors (or not) with the options `--ansi` or `--no-ansi`, respectively:

```
$ behat --no-ansi
```

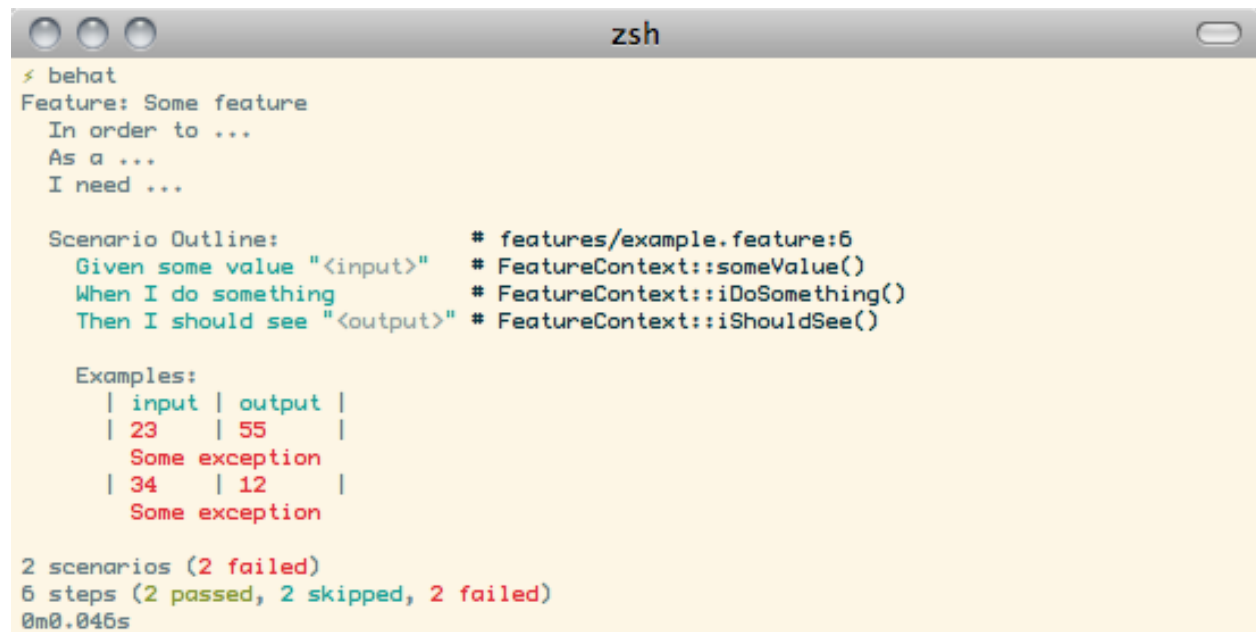
Behat prints suite execution time information after each run. If you don't want this information, you can turn it off with the `--no-time` option:

```
$ behat --no-time
```

Also, there are a bunch of options to hide some default output information from the output:

- `--no-paths` - hides paths after steps and scenarios.
- `--no-snippets` - hides snippet proposals for undefined steps after statistics.
- `--snippets-paths` - prints step information with snippets.
- `--no-multiline` - hides multiline arguments (tables and pystrings) from pretty output.

By default, Behat prints scenario outlines the same as you define them:



```
zsh
$ behat
Feature: Some feature
  In order to ...
  As a ...
  I need ...

Scenario Outline:
  Given some value "<input>"      # features/example.feature:6
  When I do something             # FeatureContext::someValue()
  Then I should see "<output>"    # FeatureContext::iDoSomething()
                                # FeatureContext::iShouldSee()

Examples:
  | input | output |
  | 23    | 55     |
  |       | Some exception
  | 34    | 12     |
  |       | Some exception

2 scenarios (2 failed)
6 steps (2 passed, 2 skipped, 2 failed)
0m0.046s
```

The output is pretty minimal and enough for you to see some errors. But in some complex cases, it may be hard to actually find failed steps in the output. To make this easier, behat offers the `--expand` option:

```
$ behat --expand
```

This options will make the previous output more verbose:



```

$ behat --expand
Feature: Some feature
  In order to ...
  As a ...
  I need ...

Scenario Outline:
  Given some value "<input>"
  When I do something
  Then I should see "<output>"
# features/example.feature:6
# FeatureContext::someValue()
# FeatureContext::iDoSomething()
# FeatureContext::iShouldSee()

Examples: | 23 | 55 |
  Given some value "23"
  When I do something
  Some exception
  Then I should see "55"
# FeatureContext::someValue()
# FeatureContext::iDoSomething()
# FeatureContext::iShouldSee()

Examples: | 34 | 12 |
  Given some value "34"
  When I do something
  Some exception
  Then I should see "12"
# FeatureContext::someValue()
# FeatureContext::iDoSomething()
# FeatureContext::iShouldSee()

2 scenarios (2 failed)
6 steps (2 passed, 2 skipped, 2 failed)
0m0.048s

```

```
$ behat --name="element of feature"
```

Only execute the feature elements which match part of the given name or regex.

If the element is part of feature-definition behat executes the whole feature, otherwise one or more scenarios.

```
$ behat --rerun="return.log"
```

Save list of failed scenarios into new file or use existing file to run only scenarios from it.

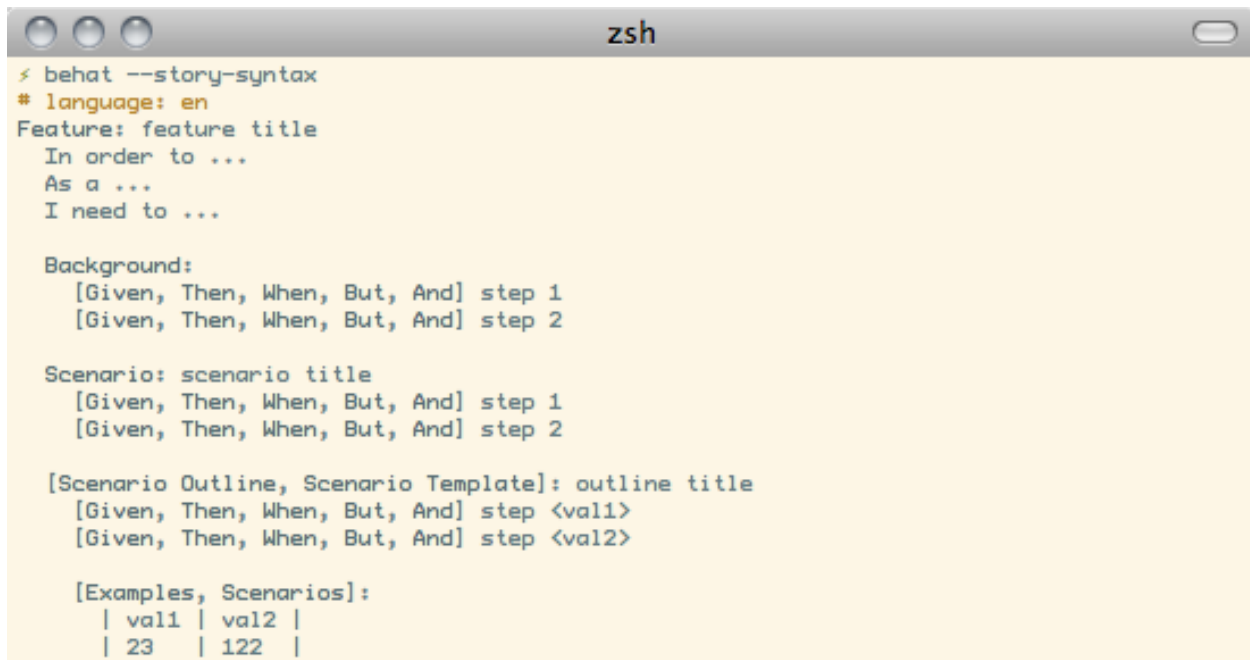
If the file contains no scenarios, then all scenarios will be executed.

## 2.6.4 Help Options

If you don't know where to start with the *Gherkin syntax*, Behat can help you with some feature example:

```
$ behat --story-syntax
```

This command will print an example feature for you to understand what keywords to use and where to use them in your \*.feature files:



```

zsh
$ behat --story-syntax
# language: en
Feature: feature title
  In order to ...
  As a ...
  I need to ...

  Background:
    [Given, Then, When, But, And] step 1
    [Given, Then, When, But, And] step 2

  Scenario: scenario title
    [Given, Then, When, But, And] step 1
    [Given, Then, When, But, And] step 2


  [Scenario Outline, Scenario Template]: outline title
    [Given, Then, When, But, And] step <val1>
    [Given, Then, When, But, And] step <val2>

  [Examples, Scenarios]:
    | val1 | val2 |
    | 23  | 122  |
  
```

If you write features in a language other than English, you can view a localized example feature in your language of choice by using the `--lang` option:

```
$ behat --story-syntax --lang fr
```

Will print the feature example in French:



```

zsh
$ behat --story-syntax --lang fr
# language: fr
Fonctionnalité: feature title
  In order to ...
  As a ...
  I need to ...

  Contexte:
    [Etant donné, Lorsqu', Lorsque, Quand, Alors, Soit, Mais, Et] step 1
    [Etant donné, Lorsqu', Lorsque, Quand, Alors, Soit, Mais, Et] step 2

  Scénario: scenario title
    [Etant donné, Lorsqu', Lorsque, Quand, Alors, Soit, Mais, Et] step 1
    [Etant donné, Lorsqu', Lorsque, Quand, Alors, Soit, Mais, Et] step 2

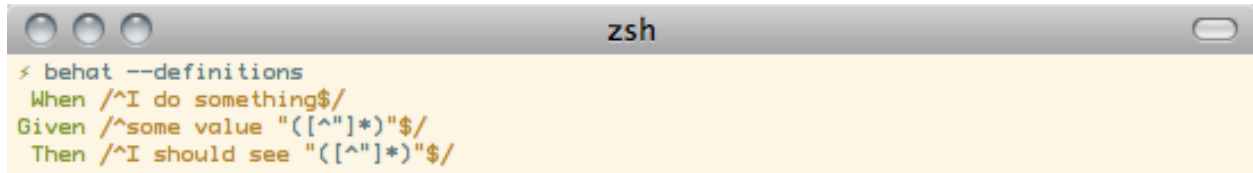
  [Plan du scénario, Plan du Scénario]: outline title
    [Etant donné, Lorsqu', Lorsque, Quand, Alors, Soit, Mais, Et] step <val1>
    [Etant donné, Lorsqu', Lorsque, Quand, Alors, Soit, Mais, Et] step <val2>

  Exemples:
    | val1 | val2 |
    | 23  | 122  |
  
```

Also, if you forgot what step definitions you've already implemented, or how to spell a particular step, behat will print all available definitions by calling it with the `-dl` option:

```
$ behat -dl
```

This command will print all available definition regular expressions:



```

$ behat --definitions
When /^I do something$/
Given /^some value "([^"]*)"$/
Then /^I should see "([^"]*)"$/

```

If you want to get extended info about definitions, use:

```
$ behat -di
```

To search for a specific definition, use:

```
$ behat -d 'search string'
```

## 2.6.5 Gherkin Filters

If you want to run only part of your suite, or some scenarios, you can do it with name or tags filters.

Tag filters supports three logical operators:

- Tags separated by commas will be combined using the OR operation
- Tags separated by && will be combined using the AND operation
- Tags preceded by ~ will be excluded using the NOT operation

For example:

```

$ behat --tags '@orm,@database'
$ behat --tags 'ticket,723'
$ behat --tags '@orm&&@fixtures'
$ behat --tags '~@javascript'
$ behat --name 'number has'

```

The first command will run only features or scenarios which have the @orm OR @database tag (or both).

The second command will run only features or scenarios which have the @ticket OR @723 tag (or both).

The third command will run only features or scenarios with both the @orm AND @fixtures tags (must have both).

The fourth command will run all features or scenarios NOT tagged with @javascript. (All except those tagged @javascript)

The fifth command will run only features and scenarios that contain number has in their title.

## 2.7 Configuration - behat.yml

Sometimes, Behat's default configuration is not enough. Some day you'll need some extended tools to configure your feature suite. For that day, Behat has a very powerful configuration system based on YAML configuration files and profiles.

### 2.7.1 behat.yml

All configuration happens inside a single configuration file in the YAML format. Behat tries to load behat.yml or config/behat.yml by default, or you can tell Behat where your config file is with the --config option:

```
$ behat --config custom-config.yml
```

All configuration parameters in that file are defined under a profile name `root` (default: for example). A profile is just a custom name you can use to quickly switch testing configuration by using the `--profile` option when executing your feature suite.

The default profile is always `default`. All other profiles inherit parameters from the `default` profile. If you only need one profile, define all of your parameters under the `default : root`:

```
# behat.yml
default:
    #...
```

### Paths

The first configuration block is `paths`. Parameters under this configuration block tell Behat where to find your feature and bootstrap files:

```
# behat.yml
default:
    paths:
        features: features
        bootstrap: %behat.paths.features%/bootstrap
```

- The `features` parameter defines where Behat will look for your `*.feature` files. The given directory will be scanned recursively.
- The `bootstrap` parameter defines the directory from which Behat will automatically load all `*.php` files.

---

**Tip:** Notice the `%behat.paths.features%` placeholder. These strings are predefined configuration variables, that you can use to build very flexible configurations.

A **variable** is a placeholder that consists of lower-case letters and starts/ends with a single `%`. These variables are your current configuration parameters, which you can use to nest configurations. Usable variables are:

1. `%behat.paths.base%` - current working dir or configuration path (if configuration file exists and loaded).
  2. `%behat.paths.features%` - features path.
  3. `%behat.paths.bootstrap%` - bootstrap path.
- 

### Filters

Another very useful configuration block is the `filters` block. This block defines default filters (name or tag) for your features. If you find yourself typing the same filters again and again from run to run, it would be more efficient for you to define them as parameters:

```
# behat.yml
default:
    filters:
        tags: "@wip"
```

These filter parameters (name and tags) accept the same strings as the Behat `--name` or `--tags` parameters do.

## Formatter

If you need to customize your output formatter, the `formatter` block is right for you:

```
# behat.yml
default:
  formatter:
    name: pretty
    parameters:
      decorated: true
      verbose: false
      time: true
      language: en
      output_path: null
      multiline_arguments: true
  #...
```

- `name` defines the default output formatter name to use for your features. You could write a class name here so Behat will use your custom class as the default output formatter, but be careful - this class should be accessible by Behat and implement `Behat\Behat\Formatter\FormatterInterface`.
- The `parameters` section defines additional parameters which will be provided into the formatter instance. As you can see, all parameters from this section duplicate `behat` tool options. You can redefine `behat` formatter defaults here. Also, this is the place to specify parameters for your custom formatters.

The YAML configuration file supports the same formatter parameters as the `behat` tool, so you can give multiple options for e.g. different formatters. This is useful when run in a continuous integration (CI) environment, so you get machine-readable output for JUnit as well as human-readable text in one single run.

An example that generates multiple output formats could look like this:

```
# behat.yml
ci:
  formatter:
    name: pretty, junit, html
    parameters:
      output_path: null, junit, behat_report.html
```

This will write pretty text output to the console, one XML file per feature to the `junit` directory and an HTML report to the file `behat_report.html`. See each formatter's documentation for details on what parameters are available, optional or mandatory.

## Colors

New in version 2.2.

As of version 2.2, you can configure Behat formatters to use specific output styles (colors).

```
default:
  formatter:
    name: pretty
    parameters:
      output_styles:
        comment: [ black, white, [ underscore ] ]
```

this will force Behat to print comments (key of the style) with black foreground (first parameter), white background (second parameter) and as underscore (list of options - third parameter).

Styles available for redefinition:

- `undefined` - style of undefined step
- `pending` - style of pending step
- `pending_param` - style of param in pending step
- `failed` - style of failed step
- `failed_param` - style of param in failed step
- `passed` - style of passed step
- `passed_param` - style of param in passed step
- `skipped` - style of skipped step
- `skipped_param` - style of param in skipped step
- `comment` - style of comment
- `tag` - style of scenario/feature tag

Available colors for first two arguments (fg and bg) are: black, red, green, yellow, blue, magenta, cyan, white

Available options are: bold, underscore, blink, reverse, conceal

## Context

Sometimes you may want to use a different default `context` class or provide useful parameters for the context constructor from your `behat.yml`. Use the `context` block to set these options:

```
# behat.yml
default:
  context:
    class:          Your\Custom\Context
    parameters:
      base_url:     http://test.mink.loc
```

- `class` defines which class you want to use as the environment. This class should be accessible by Behat and implement `Behat\Behat\Context\ContextInterface`.
- `parameters` parameters is a simple array that will be passed into the constructor of your context class when instantiated, which happens before each scenario.

## 2.7.2 Profiles

Profiles help you define different configurations for running your feature suite. Let's say we need 2 different configurations that share common options, but use different formatters. Our `behat.yml` might look like this:

```
# behat.yml
default:
  context:
    class:          Your\Custom\Context
wip:
  filters:
    tags:           "@wip"
  formatter:
    name:           progress
ci:
  formatter:
```



```

name:      junit
parameters:
  output_path: /var/tmp/junit

```

This file defines 2 additional profiles (additional to default). Every profile will use `Your\Custom\Context` as its environment object, but the `wip` profile will run only scenarios with the `@wip` (work in progress) tag and will output them with the `progress` formatter. The `ci` profile will run all features and output them with the `junit` formatter to the `/var/tmp/junit` path.

To run each of these custom profiles, use the `--profile` option:

```

behat --profile wip
behat --profile ci

```

### 2.7.3 Extensions

The `extensions` block allows you to activate extensions for your suite or for specific profile of the suite:

```

# behat.yml
default:
  extensions:
    Behat\Symfony2Extension\Extension: ~

mink:
  extensions:
    mink-extension.phar:
      base_url: http://domain.org

api:
  extensions:
    Behat\WebApiExtension\Extension:
      base_url: http://api.domain.org

```

In the example above, we activate 2 extensions depending on profile. `mink` profile will have activate `MinkExtension` and `api` profile will have `WebApiExtension`, but both of them will also have `Symfony2Extension` activated as any profile always inherit from default profile.

Extensions help you integrate Behat with frameworks and tools, that you might need to ease your test suite building.

### 2.7.4 Imports

The `imports` block allows you to share your feature suite configuration between projects and their test suites:

```

# behat.yml
imports:
  - some_installed_pear_package_or_lib/behat.yml
  - /full/path/to/custom_behat_config.yml

```

All files from the `imports` block will be loaded by Behat and merged into your `behat.yml` config.

### 2.7.5 Environment Variable

New in version 2.2.5.

If you want to configure some system-wide Behat defaults, then `BEHAT_PARAMS` environment variable is right for you:

```
export BEHAT_PARAMS="formatter[name]=progress&context[parameters][base_url]=http://localhost"
```

You could setup default value for any option, that available for you in `behat.yml`. Just provide options in *url* format (parseable by `parse_str()` php function). Behat will use those options as default ones and you will always be able to redefine them with `project behat.yml` (it has higher priority).

Learn specific solutions for specific needs:

## 3.1 Developing Web Applications with Behat and Mink

You can use Behat to describe anything that you can describe in business logic. Tools, GUI applications, web applications, etc. The most interesting part is web applications. First, behavior-driven testing already exists in the web world - it's called *functional* or *acceptance* testing. Almost all popular frameworks and languages provide functional testing tools. Today we'll talk about how to use Behat for functional testing of web applications.

### 3.1.1 Understanding Mink

One of the most important parts in the web is a browser. A browser is the window through which web application users interact with the application and other users.

So, in order to test our web application, we should transform user actions into steps and expected outcomes - with Behat that's quite simple really. The next part is much harder - run these actions and test against the expected outcome. For example, how to programmatically do things like this:

```
Given I am on "/index.php"
```

You'll need something to simulate browser application. Scenario steps would simulate a user and the browser emulator would simulate a browser with which the user interacts in order to talk to the web application.

Now the real problem. We have 2 completely different types of solutions:

- *Headless browser emulators* - browser emulators that can be executed fully without GUI through console. Such emulators can do HTTP requests and emulate browser applications on a high level (HTTP stack), but on a lower level (JS, CSS) they are totally limited. They are much faster than real browsers, because you don't need to parse CSS or execute JS in order to open pages or click links with them.
- *In-browser emulators* - this type of emulator works with real browsers, taking full control of them and using them as zombies for its testing needs. This way, you'll have a standard, fully-configured, real browser, which you will be able to control. CSS styling, JS and AJAX execution - all supported out of the box.

The problem is we need both these emulator types in order to do successful functional testing. Both these tools are quite limited at some tasks, but succeed at others. For example, you can't use in-browser emulators for all tests in your application, because this makes your tests become very slow. Also, you can't do AJAX with a headless browser.

You should use them both. But here comes a problem - these are very different tools and they have much different APIs. Using both those APIs limits us very much and in case of Behat, this problem becomes even worse, because now you have a single:

**When** I go to `"/news.php"`

And this step should be somehow executed through one or another browser emulator at will.

Here comes Mink. Mink is a browser emulator abstraction layer. It hides emulator differences behind a single, consistent API.

Just some of the benefits:

1. Single, consistent API.
2. Almost zero configuration.
3. Support for both in-browser and headless browser emulators.

### 3.1.2 Installing Mink

Mink is a PHP 5.3+ library that you'll use inside your test and feature suites. Before you begin, ensure that you have at least PHP 5.3.1 installed.

Mink integration into Behat happens thanks to MinkExtension. The extension takes care of all configuration and initialization of the Mink, leaving only the fun parts to you.

Mink should be installed through Composer.

Create `composer.json` file in the project root:

```
{
    "require": {
        "behat/behat": "~2.5",
        "behat/mink-extension": "~1.3",
        "behat/mink-goutte-driver": "~1.2",
        "behat/mink-selenium2-driver": "~1.2"
    },
    "config": {
        "bin-dir": "bin/"
    }
}
```

---

**Note:** Note that we also installed two Mink drivers - `goutte` and `selenium2`. That's because by default, Composer installation of Mink doesn't include any driver - you should choose what to use by yourself.

The easiest way to get started is to go with `goutte` and `selenium2` drivers, but note that there's bunch of other drivers available for Mink - read about them in Mink documentation.

---

Then download `composer.phar` and run update command:

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar update
```

After that, you will be able to run Behat with:

```
$ bin/behat -h
```

And this executable will already autoload all the needed classes in order to **activate** MinkExtension through `behat.yml` in the project root.

Now lets activate it:

```
# behat.yml
default:
  extensions:
    Behat\MinkExtension\Extension:
      goutte: ~
      selenium2: ~
```

You could check that extension is properly loaded by calling:

```
$ bin/behat -dl
```

It should show you all the predefined web steps as MinkExtension will automatically use the bundled MinkContext if no user-defined context class is found.

### MinkContext for Behat requirements

MinkExtension comes bundled with MinkContext, which will be used automatically by Behat as main context class if no user-defined context class found. That's why `behat -dl` shows you step definitions even when you haven't created a custom `FeatureContext` class or even a `features` folder.

## 3.1.3 Writing your first Web Feature

Let's write a feature to test Wikipedia search abilities:

```
# features/search.feature
Feature: Search
  In order to see a word definition
  As a website user
  I need to be able to search for a word

  Scenario: Searching for a page that does exist
    Given I am on "/wiki/Main_Page"
    When I fill in "search" with "Behavior Driven Development"
    And I press "searchButton"
    Then I should see "agile software development"

  Scenario: Searching for a page that does NOT exist
    Given I am on "/wiki/Main_Page"
    When I fill in "search" with "Glory Driven Development"
    And I press "searchButton"
    Then I should see "Search results"
```

We have two scenarios here:

- *Searching for a page that does exist* - describes how Wikipedia searches for pages that do exist in Wikipedia's index.
- *Searching for a page that does NOT exist* - describes how Wikipedia searches for pages that do not exist in Wikipedia's index.

As you might see, URLs in scenarios are relative, so we should provide the correct `base_url` option for MinkExtension in our `behat.yml`:

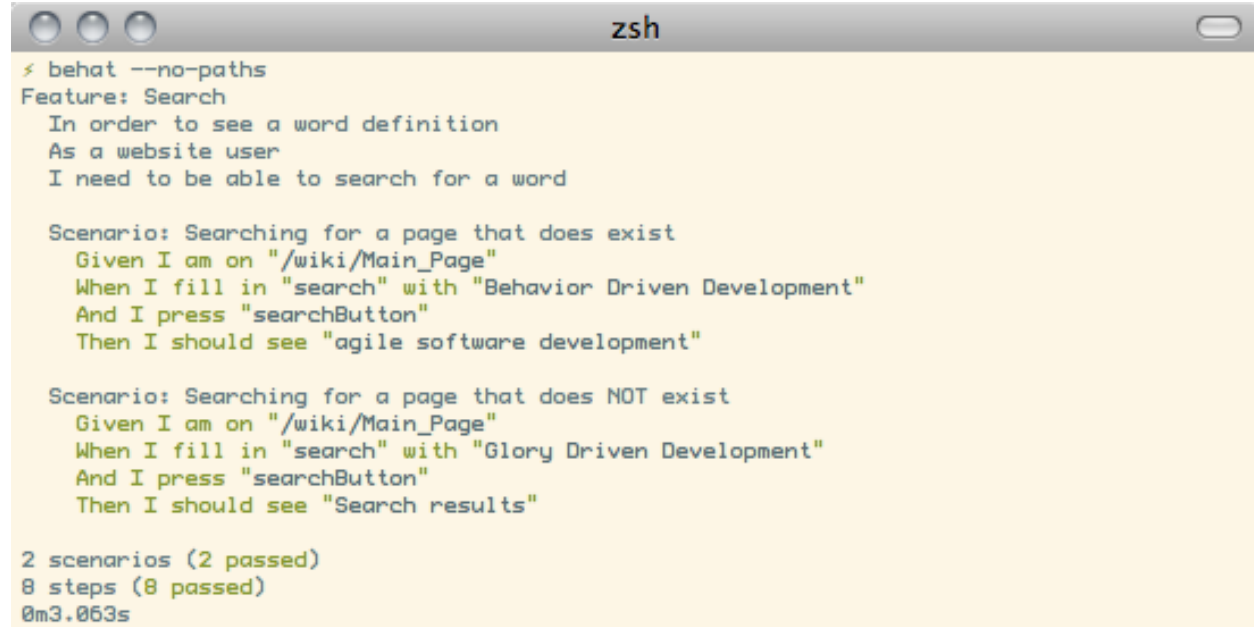
```
# behat.yml
default:
  extensions:
    Behat\MinkExtension\Extension:
```

```
base_url: http://en.wikipedia.org
goutte: ~
selenium2: ~
```

Now, run your feature:

```
$ bin/behat features/search.feature
```

You'll see output like this:

A screenshot of a zsh terminal window. The window title is 'zsh'. The terminal output shows the execution of 'bin/behat --no-paths features/search.feature'. It displays a feature 'Search' with a description 'In order to see a word definition As a website user I need to be able to search for a word'. There are two scenarios: 'Searching for a page that does exist' and 'Searching for a page that does NOT exist'. The first scenario passes with steps: 'Given I am on "/wiki/Main\_Page"', 'When I fill in "search" with "Behavior Driven Development"', 'And I press "searchButton"', and 'Then I should see "agile software development"'. The second scenario also passes with steps: 'Given I am on "/wiki/Main\_Page"', 'When I fill in "search" with "Glory Driven Development"', 'And I press "searchButton"', and 'Then I should see "Search results"'. The final summary shows '2 scenarios (2 passed)', '8 steps (8 passed)', and a duration of '0m3.063s'.

### 3.1.4 Test In-Browser - *selenium2* Session

OK. We've successfully described Wikipedia search and Behat tested it flawlessly. But what about search field autocompletion? It's done using JS and AJAX, so we can't use the default headless session to test it - we need a `javascript` session and Selenium2 browser emulator for that task.

Selenium2 gives you the ability to take full control of a real browser with a clean consistent proxy API. And Mink uses this API extensively in order to use the same Mink API and steps to do **real** actions in a **real** browser.

All you need to do is install Selenium:

1. Download latest Selenium jar from the: <http://seleniumhq.org/download/>
2. Run Selenium2 jar before your test suites (you can start this proxy during system startup):

```
java -jar selenium-server-*.jar
```

That's it. Now you should create a specific scenario in order for it to be runnable through Selenium:

```
Scenario: Searching for a page with autocompletion
Given I am on "/wiki/Main_Page"
When I fill in "search" with "Behavior Driv"
And I wait for the suggestion box to appear
Then I should see "Behavior Driven Development"
```

Now, we need to tell Behat and Mink to run this scenario in a different session (with a different browser emulator). Mink comes with a special *hook*, that searches `@javascript` or `@mink:selenium2` tag before scenario and switches the current Mink session to Selenium2 (in both cases). So, let's simply add this tag to our scenario:

```
@javascript
Scenario: Searching for a page with autocompletion
    Given I am on "/wiki/Main_Page"
    When I fill in "search" with "Behavior Driv"
    And I wait for the suggestion box to appear
    Then I should see "Behavior-driven development"
```

Now run your feature again:

```
$ bin/behat features/search.feature
```

And of course, you'll get:

That's because you have used custom `Then I wait for the suggestion box to appear` step, but have not defined it yet. In order to do that, we will need to create our own `FeatureContext` class (at last).

### 3.1.5 Defining our own `FeatureContext`

The easiest way to create context class is to ask Behat do it for you:

```
$ bin/behat --init
```

This command will create `features/bootstrap` folder and `features/bootstrap/FeatureContext.php` class for you.

Now lets try to run our feature again (just to check that everything works):

```
$ bin/behat features/search.feature
```

Oh... Now Behat tells us that all steps are undefined. What's happening there?

As we've created our own context class, MinkExtension stopped using its own bundled context class as main context and Behat uses your very own `FeatureContext` instead, which of course doesn't have those Mink steps **yet**. Let's add them.

There are multiple ways to bring the steps that are bundled with MinkExtension into your own context class. The simplest one is to use inheritance. Just extend your context from `Behat\MinkExtension\Context\MinkContext` instead of the base `BehatContext`.

Note that you will also have to do this if you've already been using Behat in your project, but without Mink, and are now adding Mink to your testing:

```
<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\TranslatedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

use Behat\MinkExtension\Context\MinkContext;

/**
```

```

* Features context.
*/
class FeatureContext extends MinkContext
{
}

```

To check that all MinkExtension steps are here again, run:

```
$ bin/behat -dl
```

If all works properly, you should see something like this:

Finally, lets add our custom wait step to context:

```

/**
 * @Then /^I wait for the suggestion box to appear$/
 */
public function iWaitForTheSuggestionBoxToAppear()
{
    $this->getSession()->wait(5000,
        "$(' .suggestions-results').children().length > 0"
    );
}

```

That simple. We get the current session and send a JS command to wait (sleep) for 5 seconds or until the expression in the second argument returns true. The second argument is a simple jQuery instruction.

Run the feature again and:



```

$ behat features/search.feature --no-paths
Feature: Search
  In order to see a word definition
  As a website user
  I need to be able to search for a word

  Scenario: Searching for a page that does exist
    Given I am on "/wiki/Main_Page"
    When I fill in "search" with "Behavior Driven Development"
    And I press "searchButton"
    Then I should see "agile software development"

  Scenario: Searching for a page that does NOT exist
    Given I am on "/wiki/Main_Page"
    When I fill in "search" with "Glory Driven Development"
    And I press "searchButton"
    Then I should see "Search results"

  @javascript
  Scenario: Searching for a page with autocompletion
    Given I am on "/wiki/Main_Page"
    When I fill in "search" with "Behavior Driv"
    And I wait for the suggestion box to appear
    Then I should see "Behavior Driven Development"

3 scenarios (3 passed)
12 steps (12 passed)
0m9.893s

```

Voilà!

**Tip:** Context isolation is a very important thing in functional tests. But restarting the browser after each scenario could slow down your feature suite very much. So by default, Mink tries hard to reset your browser session without reloading it (cleans all domain cookies).

In some cases it might not be enough (when you use http-only cookies for example). In that case, just add an `@insulated` tag to your scenario. The browser in this case will be fully reloaded and cleaned (before scenario):

**Feature:** Some feature with insulated scenario

```

@javascript @insulated
Scenario: isolated scenario
#...

```

### 3.1.6 Going further

Read more cookbook articles on Behat and Mink interactions:

- *Using the Symfony2 Profiler with Behat and Symfony2Extension*
- *Intercepting the redirection with Behat and Mink*

## 3.2 Migrating from Behat 1.x to 2.0

Behat 2.0 brings completely different way to handle testing part of your features. In 1.x we had 4 separate entities: environment, step definitions, hooks and bootstrap scripts. In 2.0 we have only one - Contexts. That's the biggest and

the coolest change since 1.x. It made features suites much cleaner and extensible.

There were less than half-year between 1.0 and 2.0 releases? Some users already have big feature suites and don't want to rewrite them once again. For such users, Behat 2.0 can become fully backward compatible with 3 very small steps.

### 3.2.1 Migrating Environment

There's no such things as environment or environment configuration in Behat2. But *FeatureContext* can successfully emulate environment objects from Behat 1.x. Let's say, we have next `env.php` configuration:

```
<?php features/support/env.php

$world->someInitialVar = 'initial-val';
$world->closureFunc = function() {
    // do something
};
```

The easiest way to migrate is to move this code into `FeatureContext` class:

```
<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

class FeatureContext extends BehatContext
{
    public $someInitialVar = 'initial-val';

    public function closureFunc()
    {
        // do something
    }
}
```

As you might see, your `someInitialVar` become an instance variable and `closureFunc()` just an instance method. You should move all your variables and methods carefully, changing all `$world` to `$this` in closure methods.

It might be very hard and annoying work, especially on large projects. So, as you might expect, you have another option:

```
<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

class FeatureContext extends BehatContext
{
    public $parameters = array();

    public function __construct(array $parameters)
```

```

{
    $this->parameters = $parameters;

    if (file_exists($env = __DIR__.'../../support/env.php')) {
        $world = $this;
        require_once($env);
    }
}

public function __call($name, array $args) {
    if (isset($this->$name) && is_callable($this->$name)) {
        return call_user_func_array($this->$name, $args);
    } else {
        $trace = debug_backtrace();
        trigger_error(
            'Call to undefined method ' . get_class($this) . '::' . $name .
            ' in ' . $trace[0]['file'] .
            ' on line ' . $trace[0]['line'],
            E_USER_ERROR
        );
    }
}
}

```

With this context, you'll be able to use your old `env.php` totally untouched. That's it. Full BC with 1.x environment.

### 3.2.2 Migrating Bootstrap Scripts

Now, what about `bootstrap.php`? Same story. You either move all your code into `features/bootstrap/FeatureContext.php` file right before class:

```

<?php

...

// require and load something here

class FeatureContext extends BehatContext
...

```

or you can leave `bootstrap.php` untouched and just tell `FeatureContext.php` to load it by itself:

```

<?php

...

if (file_exists($boot = __DIR__.'../../support/bootstrap.php')) {
    require_once($boot);
}

class FeatureContext extends BehatContext
...

```

That's it.

### 3.2.3 Migrating Step Definitions and Hooks

That was a hard part. Yep, you've heard me right. Closed step definitions and hooks support is much more easier to achieve, thanks to bundled with Behat2 closed loader.

The only thing, you need to do is to implement this interface with your `FeatureContext`:

```
<?php

namespace Behat\Behat\Context;

interface ClosedContextInterface extends ContextInterface
{
    function getStepDefinitionResources();
    function getHookDefinitionResources();
}
```

There's only two methods in this interface:

- `getStepDefinitionResources()` should return array of \*.php paths, that will be used as step definition resources.
- `getHookDefinitionResources()` should return array of \*.php paths, that will be used as hook definition resources.

For example, put next code in your `FeatureContext`:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\BehatContext;

/**
 * Features context.
 */
class FeatureContext extends BehatContext implements ClosedContextInterface
{
    public function getStepDefinitionResources()
    {
        return array(__DIR__ . '/../steps/basic_steps.php');
    }

    public function getHookDefinitionResources()
    {
        return array(__DIR__ . '/../support/hooks.php');
    }
}
```

Now, Behat will try to load all *step definitions* from out the `features/steps/basic_steps.php` file and *hooks* from out the `features/support/hooks.php`.

That's quite simple. But what if you have more than one definition file? Adding all this file into array by hands can become tedious. But you always can use `glob()`:

```
# features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\BehatContext;
```

```

/**
 * Features context.
 */
class FeatureContext extends BehatContext implements ClosedContextInterface
{
    public function getStepDefinitionResources()
    {
        return glob(__DIR__ . '/../steps/*.php');
    }

    public function getHookDefinitionResources()
    {
        return array(__DIR__ . '/../support/hooks.php');
    }
}

```

Yep. We will load all features/steps/\*.php files automatically. Same as this were done in Behat 1.x.

### 3.2.4 Fully BC Context

Taking all previously said into account, fully backward-compatible context will look like this:

```

<?php

use Behat\Behat\Context\ClosedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

if (file_exists(__DIR__ . '/../support/bootstrap.php')) {
    require_once __DIR__ . '/../support/bootstrap.php';
}

class FeatureContext extends BehatContext implements ClosedContextInterface
{
    public $parameters = array();

    public function __construct(array $parameters) {
        $this->parameters = $parameters;

        if (file_exists(__DIR__ . '/../support/env.php')) {
            $world = $this;
            require(__DIR__ . '/../support/env.php');
        }
    }

    public function getStepDefinitionResources() {
        if (file_exists(__DIR__ . '/../steps')) {
            return glob(__DIR__ . '/../steps/*.php');
        }
        return array();
    }

    public function getHookDefinitionResources() {
        if (file_exists(__DIR__ . '/../support/hooks.php')) {
            return array(__DIR__ . '/../support/hooks.php');
        }
    }
}

```

```
    }  
    return array();  
}  
  
public function __call($name, array $args) {  
    if (isset($this->$name) && is_callable($this->$name)) {  
        return call_user_func_array($this->$name, $args);  
    } else {  
        $trace = debug_backtrace();  
        trigger_error(  
            'Call to undefined method ' . get_class($this) . '::' . $name .  
            ' in ' . $trace[0]['file'] .  
            ' on line ' . $trace[0]['line'],  
            E_USER_ERROR  
        );  
    }  
}
```

You can just copy'n'paste this code into your `features/bootstrap/FeatureContext.php` and Behat2 will magically start to work with your 1.x feature suite.

## 3.3 Using Spin Functions for Slow Tests

Often, especially when using Mink to test web applications, you will find that Behat goes faster than your web application can keep up - it will try and click links or perform actions before the page has had chance to load, and therefore result in a failing test, that would have otherwise passed.

To alleviate this problem, we can use spin functions, to repeatedly try and action or test a condition, until it works. This article looks at applying it to Mink, but the technique is applicable to any test using Behat.

### 3.3.1 Using closures

PHP 5.3 introduced closures - anonymous functions (functions without a name) that can be passed as function arguments. This is very useful for our purpose as it allows us to pass a function that tests a certain condition, or tries to perform a certain action.

You can read more about closures in the *PHP documentation* <<http://php.net/manual/en/functions.anonymous.php>>.

### 3.3.2 Spin method

Lets start by implementing a spin method in our `FeatureContext.php`.

```
public function spin ($lambda)  
{  
    while (true)  
    {  
        try {  
            if ($lambda($this)) {  
                return true;  
            }  
        } catch (Exception $e) {  
            // do nothing  
        }  
    }  
}
```

```

        sleep(1);
    }
}

```

This will create a loop, calling our anonymous function every second, until it returns true. To allow us to access the `FeatureContext` object, we'll pass it into the function as a parameter.

### 3.3.3 Using the method

Now we have implemented it, we can make use of it in our step definitions. For our first example, let's say we're using Mink, and we want to wait until a specific element becomes visible.

The method expects us to pass it a function that returns true when the conditions are satisfied. So let's implement that.

```

$this->spin(function($context) {
    return ($context->getSession()->getPage()->findById('example')->isVisible());
});

```

This function will return whether `#example` element is visible or not. Our `spin` method will then try to run this function every second until the function returns true.

Because our `spin` method will also catch exceptions, we can also try actions that would normally throw an exception and cause the test to fail.

```

$this->spin(function($context) {
    $context->getSession()->getPage()->findById('example')->click();
    return true;
});

```

If the `#example` element isn't available to click yet, the function will throw an exception, and this will be caught by the try catch block in our `spin` method, and tried again in one second.

If no exception is thrown, the method will continue to run through and return true at the end.

---

**Note:** It is important to remember to return true at the end of the function, otherwise our `spin` method will continue to try running the function.

---

### 3.3.4 Adding a timeout

This works well when we just need to wait a while for some action to become available, but what if things have actually gone wrong? The method would just sit spinning forever. To resolve this, we can add a timeout.

```

public function spin ($lambda, $wait = 60)
{
    for ($i = 0; $i < $wait; $i++)
    {
        try {
            if ($lambda($this)) {
                return true;
            }
        } catch (Exception $e) {
            // do nothing
        }

        sleep(1);
    }
}

```

```

        $backtrace = debug_backtrace();

        throw new Exception(
            "Timeout thrown by " . $backtrace[1]['class'] . "::<" . $backtrace[1]['function'] . "()\n" .
            $backtrace[1]['file'] . ", line " . $backtrace[1]['line']
        );
    }

```

Now, if the function still isn't returning true after a minute, we will throw an exception stating where the test timed out.

### 3.3.5 Further reading

- *How to Lose Races and Win at Selenium* <<http://saucelabs.com/index.php/2011/04/how-to-lose-races-and-win-at-selenium/>>

## 3.4 Using the Symfony2 Profiler with Behat and Symfony2Extension

Accessing the Symfony2 profiler can be useful to test some parts of your web application that does not hit the browser. Assuming it is supposed to send an email, you can use the profiler to test that the email is sent correctly.

Your goal here will be to implement a step like this:

```

Then I should get an email on "stof@example.org" with:
    """
    To finish validating your account - please visit
    """

```

### 3.4.1 Bootstrapping the Email Step

First, let's implement a profiler retrieving function which will check that the current driver is the profilable one (useful when you let someone else write the features with this step to avoid misuses) and that the profiler is enabled:

```

<?php

namespace Acme\DemoBundle\Features\Context;

use Behat\Gherkin\Node\PyStringNode;
use Behat\Behat\Exception\PendingException;
use Behat\BehatBundle\Context\MinkContext;

use Behat\Mink\Exception\UnsupportedDriverActionException,
    Behat\Mink\Exception\ExpectationException;
use Behat\Symfony2Extension\Driver\KernelDriver;

use PHPUnit_Framework_ExpectationFailedException as AssertException;

/**
 * Feature context.
 */
class FeatureContext extends MinkContext
{
    public function getSymfonyProfile()

```



---

```

{
    $driver = $this->getSession()->getDriver();
    if (!$driver instanceof KernelDriver) {
        throw new UnsupportedDriverActionException(
            'You need to tag the scenario with '.
            '"@mink:symfony2". Using the profiler is not '.
            'supported by %s', $driver
        );
    }

    $profile = $driver->getClient()->getProfile();
    if (false === $profile) {
        throw new \RuntimeException(
            'The profiler is disabled. Activate it by setting '.
            'framework.profiler.only_exceptions to false in '.
            'your config'
        );
    }

    return $profile;
}
}

```

---

**Note:** You can only access the profiler when using the KernelDriver which gives you access to the kernel handling the request. You will need to tag your scenario so that the *symfony2* session is used.

@mink:symfony2

Scenario: I should receive an email

---

### 3.4.2 Implementing Email Step Logic

It is now time to use the profiler to implement our email checking step:

```

/**
 * @Given /^I should get an email on "(?P<email>[^\"]+)" with:$/
 */
public function iShouldGetAnEmail($email, PyStringNode $text)
{
    $error      = sprintf('No message sent to "%s"', $email);
    $profile    = $this->getSymfonyProfile();
    $collector  = $profile->getCollector('swiftmailer');

    foreach ($collector->getMessages() as $message) {
        // Checking the recipient email and the X-Swift-To
        // header to handle the RedirectingPlugin.
        // If the recipient is not the expected one, check
        // the next mail.
        $correctRecipient = array_key_exists(
            $email, $message->getTo()
        );
        $headers = $message->getHeaders();
        $correctXToHeader = false;
        if ($headers->has('X-Swift-To')) {
            $correctXToHeader = array_key_exists($email,
                $headers->get('X-Swift-To')->getFieldBodyModel()
            );
        }
    }
}

```

```
    }

    if (!$correctRecipient && !$correctXToHeader) {
        continue;
    }

    try {
        // checking the content
        return assertContains(
            $text->getRaw(), $message->getBody()
        );
    } catch (AssertException $e) {
        $error = sprintf(
            'An email has been found for "%s" but without '.
            'the text "%s".', $email, $text->getRaw()
        );
    }

    throw new ExpectationException($error, $this->getSession());
}
```

## 3.5 Intercepting the redirection with Behat and Mink

Intercepting a redirection to execute some steps before following it can be useful in some cases, for instance when you are redirecting after sending an email and want to test it *using the profiler*. This is possible for drivers based on the Symfony BrowserKit component.

### 3.5.1 Adding the Needed Steps

To intercept the redirections, you will need two new steps: one to enable the interception (allowing you to intercept the redirection for a step), and another one to follow the redirection manually when you are ready and disable the interception.

A scenario using them will look like this:

```
When I submit the form without redirection
# At this place, the redirection is not followed automatically
# This allows using the profiler for this request
Then I should receive an email
# The redirection can then be followed manually
And I should be redirected
# The driver uses the normal behavior again after this
```

### 3.5.2 Bootstrapping the Interception Steps:

First, let's implement a function which will check that the current driver is able to intercept the redirections (useful when you let someone else write the features with this step to avoid misuses):

```
<?php

namespace Acme\DemoBundle\Features\Context;
```

---

```

use Behat\BehatBundle\Context\MinkContext;
use Behat\Behat\Context\Step;
use Behat\Mink\Exception\UnsupportedDriverActionException;
use Behat\Mink\Driver\GoutteDriver;

/**
 * Feature context.
 */
class FeatureContext extends MinkContext
{
    public function canIntercept()
    {
        $driver = $this->getSession()->getDriver();
        if (!$driver instanceof GoutteDriver) {
            throw new UnsupportedDriverActionException(
                'You need to tag the scenario with ' .
                '"@mink:goutte" or "@mink:symfony2". ' .
                'Intercepting the redirections is not ' .
                'supported by %s', $driver
            );
        }
    }
}

```

---

**Note:** You can only intercept the redirections when using the GoutteDriver or the SymfonyDriver which are based on the Symfony BrowserKit component. You will need to tag your scenario so that the *goutte* or the *symfony* session is used.

@mink:symfony2

Scenario: I should receive an email

---

### 3.5.3 Implementing Interception Steps Logic

It is now time to use the client to configure the interception:

```

/**
 * @Given /^(.*) without redirection$/
 */
public function theRedirectionsAreIntercepted($step)
{
    $this->canIntercept();
    $this->getSession()->getDriver()->getClient()->followRedirects(false);

    return new Step\Given($step);
}

/**
 * @When /^I follow the redirection$/
 * @Then /^I should be redirected$/
 */
public function iFollowTheRedirection()
{
    $this->canIntercept();
    $client = $this->getSession()->getDriver()->getClient();
    $client->followRedirects(true);
    $client->followRedirect();
}

```

}

---

## Useful Resources

---

Other useful resources for learning/using Behat:

- [Cheat Sheet](#) - Behat and Mink Cheat Sheets by Jean-François Lépine
- [Behat API](#) - Behat code API
- [Gherkin API](#) - Gherkin parser API



---

## **More about Behavior Driven Development**

---

Once you're up and running with Behat, you can learn more about behavior-driven development via the following links. Though both tutorials are specific to Cucumber, Behat shares a lot with Cucumber and the philosophies are one and the same.

- [Dan North's "What's in a Story?"](#)
- [Cucumber's "Backgrounder"](#)