Behat

Release 3.0.12

January 18, 2016

Contents

1 Introduction **3**
1.1 Behaviour Driven Development . . . 3

2 Quick Start **5**
2.1 Example . . . 5
2.2 Installation . . . 7
2.3 Development . . . 8

3 User Guide **17**
3.1 About Gherkin Language . . . 17
3.2 Features and Scenarios . . . 18
3.3 Initializing a New Behat Project . . . 19
3.4 Writing Scenarios . . . 20
3.5 Organizing Features and Scenarios . . . 26
3.6 Feature Contexts . . . 26
3.7 Command Line Tool . . . 45
3.8 Configuration . . . 47

4 Cookbooks **51**
4.1 Integrating Symfony2 with Behat . . . 51
4.2 Gathering Contexts when using Multiple Contexts . . . 53

5 Useful Resources **55**
5.1 Integrating Behat with PHPStorm . . . 55
5.2 Behat cheat sheet . . . 55

Behat is an open source Behavior Driven Development framework for PHP 5.3+. What's *behavior driven development*, you ask? It's a way to develop software through a constant communication with stakeholders in form of examples; examples of how this software should help them, and you, to achieve your goals.

For example, imagine you're about to create the famous UNIX `ls` command. Before you begin, you will have a conversation with your stakeholders (UNIX users) and they might say that even though they like UNIX a lot, they need a way to see all the files in the current working directory. You then have a back-and-forth chat with them about how they see this feature working and you come up with your first scenario (an alternative name for example in BDD methodology):

```
Feature: Listing command
  In order to change the structure of the folder I am currently in
  As a UNIX user
  I need to be able see the currently available files and folders there

  Scenario: Listing two 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
      """
```

If you are a stakeholder, this is your proof that developers understand exactly how you want this feature to work. If you are a developer, this is your proof that the stakeholder expects you to implement this feature exactly in the way you're planning to implement it.

So, as a developer your work is done as soon as you've made the `ls` command, and made it behave as described in the "Listing command" scenario.

You've probably heard about this modern development practice called TDD, where you write tests for your code before, not after, the code. Well, BDD is like that, except that you don't need to come up with a test - your *scenarios* are your tests. 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 heavily inspired by Ruby's Cucumber project. Since v3.0, Behat is considered an official Cucumber implementation in PHP and is part of one big family of BDD tools.

CHAPTER 1

Introduction

1.1 Behaviour Driven Development

Once you're up and running with Behat, you can learn more about behaviour 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"

CHAPTER 2

Quick Start

Welcome to Behat! Behat is a tool to close the Behavior Driven Development (BDD) communication loop. BDD is a methodology for developing software through continuous example-based communication between developers and a business, which this application supports. This communication happens in a form that both the business and developers can clearly understand - examples. Examples are structured around the `Context-Action-Outcome` pattern and are written in a special format called *Gherkin*. The fact that Gherkin is very structural makes it very easy to automate and autotest your behaviour examples against a developing application. Automated examples are then actually used to drive this application development TDD-style.

To become a *Behat'er* in 30 minutes, just dive into the quick-start guide and enjoy!

2.1 Example

Let's imagine that you are building a completely new e-commerce platform. One of the key features of any online shopping platform is the ability to buy products. But before buying anything, customers should be able to tell the system which products they are interested in buying. You need a basket. So let's write our first user-story:

```
Feature: Product basket
  In order to buy products
  As a customer
  I need to be able to put interesting products into a basket
```

Note: This is a basic Gherkin feature and it is a simple description of this feature's story. Every feature starts with this same format: a line with the title of the feature, followed by three lines that describe the benefit, the role and the feature itself with any amount of additional description lines following after.

Before we begin to work on this feature, we must fulfil a promise of any user-story and have a real conversation with our business stakeholders. They might say that they want customers to see not only the combined price of the products in the basket, but the price reflecting both the VAT (20%) and the delivery cost (which depends on the total price of the products):

```
Feature: Product basket
  In order to buy products
  As a customer
  I need to be able to put interesting products into a basket

  Rules:
  - VAT is 20%
  - Delivery for basket under £10 is £3
  - Delivery for basket over £10 is £2
```

So as you can see, it already becomes tricky (ambiguous at least) to talk about this feature in terms of *rules*. What does it mean to add VAT? What happens when we have two products, one of which is less than £10 and another that is more? Instead you proceed with having a back-and-forth chat with stakeholders in form of actual examples of a *customer* adding products to the basket. After some time, you will come up with your first behaviour examples (in BDD these are called *scenarios*):

```
Feature: Product basket
  In order to buy products
  As a customer
  I need to be able to put interesting products into a basket

  Rules:
  - VAT is 20%
  - Delivery for basket under £10 is £3
  - Delivery for basket over £10 is £2

  Scenario: Buying a single product under £10
    Given there is a "Sith Lord Lightsaber", which costs £5
    When I add the "Sith Lord Lightsaber" to the basket
    Then I should have 1 product in the basket
    And the overall basket price should be £9

  Scenario: Buying a single product over £10
    Given there is a "Sith Lord Lightsaber", which costs £15
    When I add the "Sith Lord Lightsaber" to the basket
    Then I should have 1 product in the basket
    And the overall basket price should be £20

  Scenario: Buying two products over £10
    Given there is a "Sith Lord Lightsaber", which costs £10
    And there is a "Jedi Lightsaber", which costs £5
    When I add the "Sith Lord Lightsaber" to the basket
    And I add the "Jedi Lightsaber" to the basket
    Then I should have 2 products in the basket
    And the overall basket price should be £20
```

Note: 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.

This is your and your stakeholders' shared understanding of the project written in a structured format. It is all

based on the clear and constructive conversation you have had together. Now you can put this text in a simple file - `features/basket.feature` - under your project directory and start implementing the feature by manually checking if it fits the defined scenarios. No tools (Behat in our case) needed. That, in essence, is what BDD is.

If you are still reading, it means you are expecting more. Good! Because even though tools are not the central piece of BDD puzzle, they do improve the entire process and add a lot of benefits on top of it. For one, tools like Behat actually do close the communication loop of the story. It means that not only you and your stakeholder can together define how your feature should work before going to implement it, BDD tools allow you to automate that behaviour check after this feature is implemented. So everybody knows when it is done and when the team can stop writing code. That, in essence, is what Behat is.

Behat is an executable that you'll run from the command line to test that your application behaves exactly as you described in your `*.feature` scenarios.

Going forward, we'll show you how Behat can be used to automate this particular basket feature as a test verifying that the application (existing or not) works as you and your stakeholders expect (according to your conversation) it to.

That's it! Behat can be used to automate anything, including web-related functionality via the Mink library.

Note: If you want to learn more about the philosophy of "Behaviour Driven Development" of your application, see What's in a Story?

Note: Behat was heavily inspired by Ruby's Cucumber project. Since v3.0, Behat is considered an official Cucumber implementation in PHP and is part of one big family of BDD tools.

2.2 Installation

Before you begin, ensure that you have at least PHP 5.3.3 installed.

2.2.1 Method #1 - Composer (the recommended one)

The official way to install Behat is through Composer. Composer is a package manager for PHP. Not only can it install Behat for you right now, it will be able to easily update you to the latest version later when one comes out. If you don't have Composer already, see the Composer documentation for instructions. After that, just go into your project directory (or create a new one) and run:

```
$ php composer.phar require --dev behat/behat=~3.0.4
```

Then you will be able to check installed Behat version using:

```
$ vendor/bin/behat -V
```

2.2.2 Method #2 - PHAR (an easy one)

An easier way to install Behat is to grab a latest `behat.phar` from the download page. Make sure that you download a `3+` release. After downloading it, just place it in your project folder (or create a new one) and check the installed version using:

```
$ php behat.phar -V
```

2.3 Development

Now we will use our newly installed Behat to automate our previously written feature under the `features/basket.feature`.

Our first step after describing the feature and installing Behat is configuring the test suite. A test suite is a key concept in Behat. Suites are a way for Behat to know where to find and how to test your application against your features. By default, Behat comes with a `default` suite, which tells Behat to search for features under the `features/` folder and test them using `FeatureContext` class. Lets initialise this suite:

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

Note: If you installed Behat via PHAR, use `php behat.phar` instead of `vendor/bin/behat` in the rest of this article.

The `--init` command tells Behat to provide you with things missing to start testing your feature. In our case - it's just a `FeatureContext` class under the `features/bootstrap/FeatureContext.php` file.

2.3.1 Executing Behat

I think we're ready to see Behat in action! Let's run it:

```
$ vendor/bin/behat
```

You should see that Behat recognised that you have 3 scenarios. Behat should also tell you that your `FeatureContext` class has missing steps and proposes step snippets for you. `FeatureContext` is your test environment. It is an object through which you will describe how you would test your application against your features. It was generated by the `--init` command and now looks like this:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    /**
     * Initializes context.
     */
    public function __construct()
    {
    }
}
```

2.3.2 Defining Steps

Finally, we got to the automation part. How does Behat knows what to do when it sees `Given there is a "Sith Lord Lightsaber", which costs £5`? You tell it. You write PHP code inside your context class (`FeatureContext` in our case) and tell Behat that this code represents a specific scenario step (via an annotation with a pattern):

```
/**
 * @Given there is a(n) :arg1, which costs £:arg2
```

```
     */
    public function thereIsAWhichCostsPs($arg1, $arg2)
    {
        throw new PendingException();
    }
```

Note: `/** ... */` is a special syntax in PHP called a doc-block. It is discoverable at runtime and used by different PHP frameworks as a way to provide additional meta-information for the classes, methods and functions. Behat uses doc-blocks for step definitions, step transformations and hooks.

`@Given there is a(n) :arg1, which costs £:arg2` above the method tells Behat that this particular method should be executed whenever Behat sees step that looks like `... there is a ..., which costs £...`. This pattern will match any of the following steps:

```
Given there is a "Sith Lord Lightsaber", which costs £5
When there is a "Sith Lord Lightsaber", which costs £10
Then there is an 'Anakin Lightsaber', which costs £10
And there is a Lightsaber, which costs £2
But there is a Lightsaber, which costs £25
```

Not only that, but Behat will capture tokens (words starting with `:`, e.g. `:arg1`) from the step and pass their value to the method as arguments:

```
// Given there is a "Sith Lord Lightsaber", which costs £5
$context->thereIsAWhichCostsPs('Sith Lord Lightsaber', '5');

// Then there is a 'Jedi Lightsaber', which costs £10
$context->thereIsAWhichCostsPs('Jedi Lightsaber', '10');

// But there is a Lightsaber, which costs £25
$context->thereIsAWhichCostsPs('Lightsaber', '25');
```

Note: If you need to define more complex matching algorithms, you can also use regular expressions:

```
/**
 * @Given /there is an? \"([^\"]+)\", which costs £([\d\.]+)/
 */
public function thereIsAWhichCostsPs($arg1, $arg2)
{
    throw new PendingException();
}
```

Those patterns could be quite powerful, but at the same time, writing them for all possible steps manually could become extremely tedious and boring. That's why Behat does it for you. Remember when you previously executed `vendor/bin/behat` you got:

```
--- FeatureContext has missing steps. Define them with these snippets:

    /**
     * @Given there is a :arg1, which costs £:arg2
     */
    public function thereIsAWhichCostsPs($arg1, $arg2)
    {
        throw new PendingException();
    }
```

Behat automatically generates snippets for missing steps and all that you need to do is copy and paste them into your

context classes. Or there is an even easier way - just run:

```
$ vendor/bin/behat --dry-run --append-snippets
```

And Behat will automatically append all the missing step methods into your `FeatureContext` class. How cool is that?

If you executed `--append-snippets`, your `FeatureContext` should look like:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    /**
     * @Given there is a :arg1, which costs £:arg2
     */
    public function thereIsAWhichCostsPs($arg1, $arg2)
    {
        throw new PendingException();
    }

    /**
     * @When I add the :arg1 to the basket
     */
    public function iAddTheToTheBasket($arg1)
    {
        throw new PendingException();
    }

    /**
     * @Then I should have :arg1 product(s) in the basket
     */
    public function iShouldHaveProductInTheBasket($arg1)
    {
        throw new PendingException();
    }

    /**
     * @Then the overall basket price should be £:arg1
     */
    public function theOverallBasketPriceShouldBePs($arg1)
    {
        throw new PendingException();
    }
}
```

Note: We have removed the constructor and grouped `I should have :arg1 product in the basket` and `I should have :arg1 products in the basket` into one `I should have :arg1 product(s) in the basket`.

2.3.3 Automating Steps

Now it is finally time to start implementing our basket feature. The approach when you use tests to drive your application development is called a Test-Driven Development (or simply TDD). With TDD you start by defining test cases for the functionality you develop, then you fill these test cases with the best-looking application code you could come up with (use your design skills and imagination).

In the case of Behat, you already have defined test cases (step definitions in your `FeatureContext`) and the only thing that is missing is that best-looking application code we could come up with to fulfil our scenario. Something like this:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Tester\Exception\PendingException;
use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    private $shelf;
    private $basket;

    public function __construct()
    {
        $this->shelf = new Shelf();
        $this->basket = new Basket($this->shelf);
    }

    /**
     * @Given there is a :product, which costs £:price
     */
    public function thereIsAWhichCostsPs($product, $price)
    {
        $this->shelf->setProductPrice($product, floatval($price));
    }

    /**
     * @When I add the :product to the basket
     */
    public function iAddTheToTheBasket($product)
    {
        $this->basket->addProduct($product);
    }

    /**
     * @Then I should have :count product(s) in the basket
     */
    public function iShouldHaveProductInTheBasket($count)
    {
        PHPUnit_Framework_Assert::assertCount(
            intval($count),
            $this->basket
        );
    }

    /**
     * @Then the overall basket price should be £:price
     */
```

```
    public function theOverallBasketPriceShouldBePs($price)
    {
        PHPUnit_Framework_Assert::assertSame(
            floatval($price),
            $this->basket->getTotalPrice()
        );
    }
}
```

As you can see, in order to test and implement our application, we introduced 2 objects - `Shelf` and `Basket`. The first is responsible for storing products and their prices, the second is responsible for the representation of our customer basket. Through appropriate step definitions we declare products' prices and add products to the basket. We then compare the state of our `Basket` object with our expectations using PHPUnit assertions.

Note: Behat doesn't come with its own assertion tool, but you can use any proper assertion tool out there. A proper assertion tool is a library whose assertions throw exceptions on failure. For example, if you're familiar with PHPUnit you can use its assertions in Behat by installing it via composer:

```
$ php composer.phar require --dev phpunit/phpunit='~4.1.0'
```

and then by simply using assertions in your steps:

```
PHPUnit_Framework_Assert::assertCount(
    intval($count),
    $this->basket
);
```

Now try to execute your feature tests:

```
$ vendor/bin/behat
```

You should see a beginning of the feature and then an error saying that class `Shelf` does not exist. It means we're ready to start writing actual application code!

2.3.4 Implementing the Feature

So now we have 2 very important things:

1. A concrete user-aimed description of functionality we're trying to deliver.
2. Set of failing tests that tell us what to do next.

Now is the easiest part of application development - feature implementation. Yes, with TDD and BDD implementation becomes a routine task, because you already did most of the job in the previous phases - you wrote tests, you came up with an elegant solution (as far as you could go in current context) and you chose the actors (objects) and actions (methods) that are involved. Now it's time to write a bunch of PHP keywords to glue it all together. Tools like Behat, when used in the right way, will help you to write this phase by giving you a simple set of instructions that you need to follow. You did your thinking and design, now it's time to sit back, run the tool and follow its instructions in order to write your production code.

Lets start! Run:

```
$ vendor/bin/behat
```

Behat will try to test your application with `FeatureContext` but will fail soon, producing something like this onto your screen:

```
Fatal error: Class 'Shelf' not found
```

Now our job is to reinterpret this phrase into an actionable instruction. Like "Create the `Shelf` class". Let's go and create it inside `features/bootstrap`:

```php
// features/bootstrap/Shelf.php

final class Shelf
{
}
```

Note: We put the `Shelf` class into `features/bootstrap/Shelf.php` because `features/bootstrap` is an autoloading folder for Behat. Behat has a built-in PSR-0 autoloader, which looks into `features/bootstrap`. If you're developing your own application, you probably would want to put classes into a place appropriate for your app.

Let's run Behat again:

```
$ vendor/bin/behat
```

We will get different message on our screen:

```
Fatal error: Class 'Basket' not found
```

Good, we are progressing! Reinterpreting the message as, "Create the `Basket` class". Let's follow our new instruction:

```php
// features/bootstrap/Basket.php

final class Basket
{
}
```

Run Behat again:

```
$ vendor/bin/behat
```

Great! Another "instruction":

```
Call to undefined method Shelf::setProductPrice()
```

Follow these instructions step-by-step and you will end up with `Shelf` class looking like this:

```php
// features/bootstrap/Shelf.php

final class Shelf
{
    private $priceMap = array();

    public function setProductPrice($product, $price)
    {
        $this->priceMap[$product] = $price;
    }

    public function getProductPrice($product)
    {
        return $this->priceMap[$product];
    }
}
```

and `Basket` class looking like this:

```
// features/bootstrap/Basket.php

final class Basket implements \Countable
{
    private $shelf;
    private $products;
    private $productsPrice = 0.0;

    public function __construct(Shelf $shelf)
    {
        $this->shelf = $shelf;
    }

    public function addProduct($product)
    {
        $this->products[] = $product;
        $this->productsPrice += $this->shelf->getProductPrice($product);
    }

    public function getTotalPrice()
    {
        return $this->productsPrice
            + ($this->productsPrice * 0.2)
            + ($this->productsPrice > 10 ? 2.0 : 3.0);
    }

    public function count()
    {
        return count($this->products);
    }
}
```

Run Behat again:

```
$ vendor/bin/behat
```

All scenarios should pass now! Congratulations, you almost finished your first feature. The last step is to *refactor*. Look at the `Basket` and `Shelf` classes and try to find a way to make their code even more cleaner, easier to read and concise.

Tip: I would recommend starting from `Basket::getTotalPrice()` method and extracting VAT and delivery cost calculation in private methods.

After refactoring is done, you will have:

1. Clearly designed and obvious code that does exactly the thing it should do without any gold plating.
2. A regression test suite that will help you to be confident in your code going forward.
3. Living documentation for the behaviour of your code that will live, evolve and die together with your code.
4. An incredible level of confidence in your code. Not only are you confident now that it does exactly what it's supposed to do, you are confident that it does so by delivering value to the final users (customers in our case).

There are many more benefits to BDD but those are the key reasons why most BDD practitioners do BDD in Ruby, .Net, Java, Python and JS. Welcome to the family!

2.3.5 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.

CHAPTER 3

User Guide

3.1 About Gherkin Language

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

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

Tip: If you're still new to Behat, jump into the *Quick Start* first, then return here to learn more about Gherkin.

3.1.1 Gherkin Syntax

Like YAML and Python, Gherkin is a whitespace-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

  Additional text...

  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*".

2. The next three lines (`In order to ...`, `As an ...`, `I want to ...`) provide context to the people reading your feature and describe the business value derived from the inclusion of the feature in your software. These lines are not parsed by Behat and don't have a required structure.
3. `Scenario: Some determinable business situation` starts the scenario and contains a description of the scenario. Learn more about "*Scenarios*".
4. The next 7 lines are the scenario steps, each of which is matched to a pattern defined elsewhere. Learn more about "*Steps*".
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 pattern, which executes a PHP callback function. You can read more about steps matching and execution in "*Defining Step Definitions*".

3.1.2 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 also gives Behat the ability to have multilanguage features in one suite.

3.2 Features and Scenarios

3.2.1 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 a localized version of one of these). 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*, features may contain *Scenario Outlines* and *Backgrounds*.

3.2.2 Scenarios

Scenarios are one of the core Gherkin structures. Every scenario starts with the `Scenario:` keyword (or localized keyword), 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."
```

3.3 Initializing a New Behat Project

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

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

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

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

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\SnippetAcceptingContext;
use Behat\Gherkin\Node\PyStringNode;
use Behat\Gherkin\Node\TableNode;

class FeatureContext implements SnippetAcceptingContext
{
    /**
     * Initializes context.
     */
    public function __construct()
    {
    }
}
```

All *step definitions* and *Hooks* necessary for testing your project against your features will be represented as methods inside this class.

3.3.1 Suite Initialisation

Suites are a core part of Behat. Any feature of Behat knows about them and can give you a hand with them. For example, if you defined your suites in `behat.yml` before running `--init`, it will actually create the folders and suites you configured, instead of the default ones.

3.4 Writing Scenarios

3.4.1 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 the **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.

Given Examples

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).

Using Givens as Data Fixtures

If you use ORMs like Doctrine or Propel, we recommend using a Given step with a tables argument 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).

When Examples

Two good examples of using **Whens** 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).

Then Examples

Two good examples of using **Thens** are:

- Verify that something related to the Given + When is (or is not) in the output:

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

- Check that some external system has received the expected message:

```
When I send an email with:
  """
  ...
  """
Then the client should receive the email with:
  """
  ...
  """
```

Caution: 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 another thing
  Given yet another 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 another thing
  And yet another 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 differentiate between them - you should!

3.4.2 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*.

```
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."
```

3.4.3 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 (except for the first header row).

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: Eating
  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: Eating
  # <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
```

3.4.4 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   |
```

Attention: Don't confuse tables with scenario outlines - syntactically they are identical, but they have a different purpose. Outlines declare multiple different values for the same scenario, while tables are used to expect a set of data.

Matching Tables in your Step Definition

A matching definition for this step looks like this:

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

// ...

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

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()`).

3.4.5 Multiline Arguments

The one line steps let Behat extract 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 designed 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.

3.4.6 Pystrings

Multiline Strings (also known as PyStrings) are useful for specifying a larger piece of text. The text should be offset by delimiters consisting of three double-quote marks (`"""`), placed on their own line:

```
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.

Matching PyStrings in your Step Definition

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

```
use Behat\Gherkin\Node\PyStringNode;

// ...

/**
 * @Given a blog post named :title with:
 */
public function blogPost($title, PyStringNode $markdown)
{
    $this->createPost($title, $markdown->getRaw());
}
```

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.

3.5 Organizing Features and Scenarios

3.5.1 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.

3.6 Feature Contexts

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.

Context classes are a keystone of testing environment in Behat. The context class is a simple POPO (Plain Old PHP Object) that tells Behat how to test your features. If `*.feature` files are all about describing *how* your application behaves, then the context class is all about how to test it.

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    public function __construct($parameter)
    {
        // instantiate context
    }

    /** @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
    }
}
```

A simple mnemonic for context classes is "testing features *in a context*". Feature descriptions tend to be very high level. It means there's not much technical detail exposed in them, so the way you will test those features pretty much depends on the context you test them in. That's what context classes are.

Tip: This class can be easily created by running *Behat command line tool* with the `--init` command from your project's directory. Behat is able to support you when you are creating a new project. Learn more about "*Initializing a New Behat Project*".

3.6.1 Hooking into the Test Process

You've learned *how to write step definitions* and that with *Gherkin* you can move common steps into a 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\Context;
use Behat\Testwork\Hook\Scope\BeforeSuiteScope;
use Behat\Behat\Hook\Scope\AfterScenarioScope;

class FeatureContext implements Context
{
    /**
     * @BeforeSuite
     */
     public static function prepare(BeforeSuiteScope $scope)
     {
         // prepare system for test suite
         // before it runs
     }

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

Behat Hook System

Behat provides a number of hook points which allow us to run arbitrary logic at various points in the Behat test cycle. Hooks are a lot like step definitions or transformations - they are just simple methods with special annotations inside your context classes. There is no association between where the hook is defined and which node it is run for, but you can use tagged or named hooks if you want more fine grained control.

All defined hooks are run whenever the relevant action occurs. The action tree looks something like this:

```
-- Suite #1
|   -- Feature #1
|   |   -- Scenario #1
|   |   |   -- Step #1
|   |   |   -- Step #2
|   |   -- Scenario #2
|   |       -- Step #1
|   |       -- Step #2
|   -- Feature #2
|       -- Scenario #1
|           -- Step #1
-- Suite #2
    -- Feature #1
        -- Scenario #1
            -- Step #1
```

This is a basic test cycle in Behat. There are many test suites, each of which has many features, which themselves have many scenarios with many steps. Note that when Behat actually runs, scenario outline examples are interpreted as scenarios - meaning each outline example becomes an actual scenario in this action tree.

Hooks

Hooks allow you to execute your custom code just before or just after each of these actions. Behat allows you to use the following hooks:

1. The `BeforeSuite` hook is run before any feature in the suite runs. For example, you could use this to set up the project you are testing. This hook receives an optional argument with an instance of the `Behat\Testwork\Hook\Scope\BeforeSuiteScope` class.
2. The `AfterSuite` hook is run after all features in the suite have run. This hooks is useful to dump or print some kind of statistics or tear down your application after testing. This hook receives an optional argument with an instance of the `Behat\Testwork\Hook\Scope\AfterSuiteScope` class.
3. The `BeforeFeature` hook is run before a feature runs. This hook receives an optional argument with an instance of the `Behat\Behat\Hook\Scope\BeforeFeatureScope` class.
4. The `AfterFeature` hook is run after Behat finishes executing a feature. This hook receives an optional argument with an instance of the `Behat\Behat\Hook\Scope\AfterFeatureScope` class.
5. The `BeforeScenario` hook is run before a specific scenario will run. This hook receives an optional argument with an instance of the `Behat\Behat\Hook\Scope\BeforeScenarioScope` class.
6. The `AfterScenario` hook is run after Behat finishes executing a scenario. This hook receives an optional argument with an instance of the `Behat\Behat\Hook\Scope\AfterScenarioScope` class.
7. The `BeforeStep` hook is run before a step runs. This hook receives an optional argument with an instance of the `Behat\Behat\Hook\Scope\BeforeStepScope` class.
8. The `AfterStep` hook is run after Behat finishes executing a step. This hook receives an optional argument with an instance of the `Behat\Behat\Hook\Scope\AfterStepScope` class.

You can use any of these hooks by annotating any of your methods in your context class:

```
/**
 * @BeforeSuite
 */
public static function prepare($scope)
{
    // 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 hook you want to use (e.g. `@BeforeSuite`).

Suite Hooks

Suite hooks are run outside of the scenario context. It means that your context class (e.g. `FeatureContext`) is not instantiated yet and the only way Behat can execute code in it is through the static calls. That is why suite hooks must be defined as static methods in the context class:

```
use Behat\Testwork\Hook\Scope\BeforeSuiteScope;
use Behat\Testwork\Hook\Scope\AfterSuiteScope;

/** @BeforeSuite */
public static function setup(BeforeSuiteScope $scope)
{
}

/** @AfterSuite */
```

```
public static function teardown(AfterSuiteScope $scope)
{
}
```

There are two suite hook types available:

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

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
}
```

Scenario Hooks

Scenario hooks are triggered before or after each scenario runs. These hooks are executed inside an initialized context instance, so not only could they be simple context instance methods, they will also have access to any object properties you set during your scenario:

```
use Behat\Behat\Hook\Scope\BeforeScenarioScope;
use Behat\Behat\Hook\Scope\AfterScenarioScope;

/** @BeforeScenario */
public function before(BeforeScenarioScope $scope)
{
}

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

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.

`@AfterScenario` functions exactly the same way, being executed both after usual scenarios and outline examples.

Feature Hooks

Same as suite hooks, feature hooks are ran outside of the scenario context. So same as suite hooks, your feature hooks should be defined as static methods inside your context:

```
use Behat\Behat\Hook\Scope\BeforeFeatureScope;
use Behat\Behat\Hook\Scope\AfterFeatureScope;

/** @BeforeFeature */
public static function setupFeature(BeforeFeatureScope $scope)
{
}

/** @AfterFeature */
public static function teardownFeature(AfterFeatureScope $scope)
{
}
```

There are two feature hook types available:

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

Step Hooks

Step hooks are triggered before or after each step runs. These hooks are run inside an initialized context instance, so they are just plain context instance methods in the same way as scenario hooks are:

```
use Behat\Behat\Hook\Scope\BeforeStepScope;
use Behat\Behat\Hook\Scope\AfterStepScope;

/** @BeforeStep */
public function beforeStep(BeforeStepScope $scope)
{
}

/** @AfterStep */
public function after(AfterStepScope $scope)
{
}
```

There are two step hook types available:

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

3.6.2 Defining Step Definitions

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

```
/**
 * @When I do something with :argument
 */
public function iDoSomethingWith($argument)
{
    // do something with $argument
}
```

Note: Step definitions are just normal PHP methods. They are instance methods in a special class called *FeatureContext*.

Creating Your First Step Definition

The main goal for a step definition is to be executed when Behat sees its matching step in executed scenario. However, 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 pattern matching.

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

Behat uses php-doc annotations to bind patterns to `FeatureContext` methods:

```
/**
 * @When I do something with :methodArgument
 */
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 step text pattern (e.g. `I do something with :methodArgument`).
3. All token values of the pattern (e.g. `:methodArgument`) will be captured and passed to the method argument with the same name (`$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 have probably noticed, this pattern 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 25
```

The only real difference between those steps in the eyes of Behat is the captured token text. This text will be passed to the 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. `$context->someMethod($methodArgument = 'string1');`.
2. `$context->someMethod($methodArgument = 'some other string');`.
3. `$context->someMethod($methodArgument = '25');`.

Note: A pattern can't automatically determine the datatype of its matches, so all method arguments coming from step definitions are passed as strings. Even if your pattern 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 string 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 patterns 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 :stringArgument and with :numberArgument
 */
public function someMethod($stringArgument, $numberArgument) {}
```

You can also specify alternative words and optional parts of words, like this:

```
/**
 * @When there is/are :count monster(s)
 */
public function thereAreMonsters($count) {}
```

If you need to come up with a much more complicated matching algorithm, you can always use good old regular expressions:

```
/**
 * @When /^there (?:is|are) (\d+) monsters?$/i
 */
public function thereAreMonsters($count) {}
```

Definition Snippets

You now know how to write step definitions by hand, but writing all these method stubs, annotations and patterns by hand 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:

```
Feature:
  Scenario:
    Given some step with "string" argument
    And number step with 23
```

If your context class implements `Behat\Behat\Context\SnippetAcceptingContext` interface and you test a feature with missing steps in Behat:

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

Behat will provide auto-generated snippets for your context class.

It not only generates the proper definition annotation type (`@Given`), but also a proper pattern with tokens capturing (`:arg1`, `:arg2`), method name (`someStepWithArgument()`, `numberStepWith()`) and arguments (`$arg1`, `$arg2`), all based just on the text of the step. Isn't that cool?

The only thing left for you to do is to copy these method snippets into your `FeatureContext` class and provide a useful body for them. Or even better, run behat with `--append-snippets` option:

```
$ vendor/bin/behat features/example.feature --dry-run --append-snippets
```

`--append-snippets` tells Behat to automatically add snippets inside your context class.

Note: Implementing the `SnippetAcceptingContext` interface tells Behat that your context is expecting snippets to be generated inside it. Behat will generate simple pattern snippets for you, but if regular expressions are your thing, Behat can generate them instead if you implement `Behat\Behat\Context\CustomSnippetAcceptingContext` interface instead and add `getAcceptedSnippetType()` method returning string `"regex"`:

```
public static function getAcceptedSnippetType()
{
    return 'regex';
}
```

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 any `Exception`, the step is marked as successful (green). What you return from a definition method has no effect on the passing or failing status of the definition itself.

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

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
```

```
{
    /** @Given some step with :argument1 argument */
    public function someStepWithArgument($argument1)
    {
    }

    /** @Given number step with :argument1 */
    public function numberStepWith($argument1)
    {
    }
}
```

When you run your feature, you'll see all steps passed and are marked as green. That's simply because no exceptions were thrown during their execution.

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

Tip: Enable the "posix" PHP extension in order to see colorful Behat output. Depending on your Linux, Mac OS or other Unix system it might be part of the default PHP installation or a separate `php5-posix` package.

Undefined Steps

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

Let's pretend we have an empty context class:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
}
```

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

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** (cyan).

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\Tester\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:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;
use Behat\Behat\Tester\Exception\PendingException;

class FeatureContext implements Context
{
    /** @Given some step with :argument1 argument */
    public function someStepWithArgument($argument1)
    {
        throw new PendingException('Do some string work');
    }

    /** @Given number step with :argument1 */
    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 and one step following it that is marked cyan.

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 any `Exception` (except `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 to fail.

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

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /** @Given some step with :argument1 argument */
    public function someStepWithArgument($argument1)
    {
        throw new Exception('some exception');
    }

    /** @Given number step with :argument1 */
    public function numberStepWith($argument1)
    {
```

```
    }
}
```

When you run your feature, you'll get 1 failing step that is marked red and it will be followed by 1 skipped step that is marked cyan.

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

Note: All steps within a scenario 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 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 by installing it via composer:

```
$ php composer.phar require --dev phpunit/phpunit='~4.1.0'
```

and then by simply using assertions in your steps:

```
PHPUnit_Framework_Assert::assertCount(intval($count), $this->basket);
```

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

```
$ vendor/bin/behat features/example.feature -vv
```

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**:

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:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /** @Given /^.* step with .*$/ */
    public function someStepWithArgument()
    {
    }
```

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

Executing Behat with this feature context will result in a `Ambiguous` exception being thrown.

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 pattern more than once. For example, look at the two `@Given` patterns defined in this feature context:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /** @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.

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 an 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 pattern.

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

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /**
```

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

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

Note: In the same way as with step definitions, you can use both simple patterns and regular expressions.

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:

```
// features/bootstrap/FeatureContext.php

use Behat\Behat\Context\Context;

class FeatureContext implements Context
{
    /**
     * @Transform :user
     */
    public function castUsernameToUser($user)
    {
        return new User($user);
    }

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

Table Transformation

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:

```
// features/bootstrap/FeatureContext.php

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

class FeatureContext implements Context
{
    /**
     * @Given the following users:
     */
    public function pushUsers(TableNode $usersTable)
    {
        $users = array();
        foreach ($usersTable 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:`:

```
// features/bootstrap/FeatureContext.php

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

class FeatureContext implements Context
{
    /**
     * @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. Also, they are inherently hard to debug because of their highly dynamic nature.

Tip: Behat provides a *command line option* that allows you to easily browse definitions in order to reuse them or adapt them.

3.6.3 Context Class Requirements

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

1. The context class should implement the `Behat\Behat\Context\Context` interface.
2. The context class should be called `FeatureContext`. It's a simple convention inside the Behat infrastructure. `FeatureContext` is the name of the context class for the default suite. This can easily be changed through suite configuration inside `behat.yml`.
3. The context class should be discoverable and loadable by Behat. That means you should somehow tell Behat about your class file. Behat comes with a PSR-0 autoloader out of the box and the default autoloading directory is `features/bootstrap`. That's why the default `FeatureContext` is loaded so easily by Behat. You can place your own classes under `features/bootstrap` by following the PSR-0 convention or you can even define your own custom autoloading folder via `behat.yml`.

Note: `Behat\Behat\Context\SnippetAcceptingContext` and `Behat\Behat\Context\CustomSnippetAccepti` are special versions of the `Behat\Behat\Context\Context` interface that tell Behat this context expects snippets to be generated for it.

Tip: The *Behat command line tool* has an `--init` option that will initialize a new Behat project in your directory. Learn more about *Initializing a New Behat Project*.

3.6.4 Contexts Lifetime

Your context class is initialized before each scenario is run, 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.

3.6.5 Multiple Contexts

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 of helping make your code more structured by allowing you to use multiple contexts in a single test suite.

In order to customise the list of contexts your test suite requires, you need to fine-tune the suite configuration inside `behat.yml`:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - FeatureContext
                - SecondContext
                - ThirdContext
```

The first `default` in this configuration is a name of the profile. We will discuss profiles a little bit later. Under the specific profile, we have a special `suites` section, which configures suites inside profile. We will talk about test suites in more detail in the *next chapter*, for now just keep in mind that a suite is a way to tell Behat where to find your features and how to test them. The interesting part for us now is the `contexts` section - this is an array of context class names. Behat will use the classes specified there as your feature contexts. This means that every time Behat sees a scenario in your test suite, it will:

1. Get list of all context classes from this `contexts` option.
2. Will try to initialize all these context classes into objects.
3. Will search for *step definitions* and *Hooks* in all of them.

Note: Do not forget that each of these context classes should follow all context class requirements. Specifically - they all should implement `Behat\Behat\Context\Context` interface and be autoloadable by Behat.

Basically, all contexts under `contexts` section of your `behat.yml` are the same for Behat. It will find and use the methods in them the same way it does in the default `FeatureContext`. And if you're happy with a single context class, but you don't like the name `FeatureContext`, here's how you change it:

```
# behat.yml

default:
    suites:
```

```
        default:
            contexts:
                - MyAwesomeContext
```

This configuration will tell Behat to look for `MyAwesomeContext` instead of the default `FeatureContext`.

Note: Unlike profiles, Behat will not inherit any configuration of your `default` suite. The name `default` is only used for demonstration purpose in this guide. If you have multiple suites that all should use the same context, you will have to define that specific context for every specific suite:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext
                - MyWickedContext
        suite_a:
            contexts:
                - MyAwesomeContext
                - MyWickedContext
        suite_b:
            contexts:
                - MyAwesomeContext
```

This configuration will tell Behat to look for `MyAwesomeContext` and `MyWickedContext` when testing `suite_a` and `MyAwesomeContext` when testing `suite_b`. In this example, `suite_b` will not be able to call steps that are defined in the `MyWickedContext`. As you can see, even if you are using the name `default` as the name of the suite, Behat will not inherit any configuration from this suite.

3.6.6 Context Parameters

Context classes can be very flexible depending on how far you want to go in making them dynamic. Most of us will want to make our contexts environment-independent; where should we put temporary files, which URLs will be used to access the application? These are context configuration options highly dependent on the environment you will test your features in.

We already said that context classes are just plain old PHP classes. How would you incorporate environment-dependent parameters into your PHP classes? Use *constructor arguments*:

```
// features/bootstrap/MyAwesomeContext.php

use Behat\Behat\Context\Context;

class MyAwesomeContext implements Context
{
    public function __construct($baseUrl, $tempPath)
    {
        $this->baseUrl = $baseUrl;
        $this->tempPath = $tempPath;
    }
}
```

As a matter of fact, Behat gives you ability to do just that. You can specify arguments required to instantiate your context classes through same `contexts` setting inside your `behat.yml`:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext:
                    - http://localhost:8080
                    - /var/tmp
```

Note: Note an indentation for parameters. It is significant:

```
contexts:
    - MyAwesomeContext:
        - http://localhost:8080
        - /var/tmp
```

Aligned four spaces from the context class itself.

Arguments would be passed to the `MyAwesomeContext` constructor in the order they were specified here. If you are not happy with the idea of keeping an order of arguments in your head, you can use argument names instead:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext:
                    baseUrl: http://localhost:8080
                    tempPath: /var/tmp
```

As a matter of fact, if you do, the order in which you specify these arguments becomes irrelevant:

```
# behat.yml

default:
    suites:
        default:
            contexts:
                - MyAwesomeContext:
                    tempPath: /var/tmp
                    baseUrl: http://localhost:8080
```

Taking this a step further, if your context constructor arguments are optional:

```
public function __construct($baseUrl = 'http://localhost', $tempPath = '/var/tmp')
{
    $this->baseUrl = $baseUrl;
    $this->tempPath = $tempPath;
}
```

You then can specify only the parameter that you actually need to change:

```
# behat.yml

default:
    suites:
        default:
```

```
            contexts:
                - MyAwesomeContext:
                    tempPath: /var/tmp
```

In this case, the default values would be used for other parameters.

3.6.7 Context Traits

PHP 5.4 have brought an interesting feature to the language - traits. Traits are a mechanism for code reuse in single inheritance languages like PHP. Traits are implemented as a compile-time copy-paste in PHP. That means if you put some step definitions or hooks inside a trait:

```
// features/bootstrap/ProductsDictionary.php

trait ProductsDictionary
{
    /**
     * @Given there is a(n) :arg1, which costs £:arg2
     */
    public function thereIsAWhichCostsPs($arg1, $arg2)
    {
        throw new PendingException();
    }
}
```

And then use it in your context:

```
// features/bootstrap/MyAwesomeContext.php

use Behat\Behat\Context\Context;

class MyAwesomeContext implements Context
{
    use ProductsDictionary;
}
```

It will just work as you expect it to.

3.7 Command Line Tool

3.7.1 Identifying Tests

By Suite

By default, when you run Behat it will execute all registered suites one-by-one. If you want to run a single suite instead, use the `--suite` option:

```
$ vendor/bin/behat --suite=web_features
```

3.7.2 Informative Output

Print Definitions

As your set of features will grow, there's a good chance that the amount of different steps that you'll have at your disposal to describe new scenarios will also grow.

Behat provides a command line option `--definitions` or simply `-d` to easily browse definitions in order to reuse them or adapt them (introducing new placeholders for example).

For example, when using the Mink context provided by the Mink extension, you'll have access to its step dictionary by running:

```
$ behat -di
web_features | Given /^(?:|I )am on (?:|the )homepage$/
             | Opens homepage.
             | at `Behat\MinkExtension\Context\MinkContext::iAmOnHomepage()`

web_features | When /^(?:|I )go to (?:|the )homepage$/
             | Opens homepage.
             | at `Behat\MinkExtension\Context\MinkContext::iAmOnHomepage()`

web_features | Given /^(?:|I )am on "(?P<page>[^"]+)"$/
             | Opens specified page.
             | at `Behat\MinkExtension\Context\MinkContext::visit()`

# ...
```

or, for a shorter output:

```
$ behat -dl
web_features | Given /^(?:|I )am on (?:|the )homepage$/
web_features |  When /^(?:|I )go to (?:|the )homepage$/
web_features | Given /^(?:|I )am on "(?P<page>[^"]+)"$/
web_features |  When /^(?:|I )go to "(?P<page>[^"]+)"$/
web_features |  When /^(?:|I )reload the page$/
web_features |  When /^(?:|I )move backward one page$/
web_features |  When /^(?:|I )move forward one page$/
# ...
```

You can also search for a specific pattern by running:

```
$ behat --definitions="field" (or simply behat -dfield)
web_features | When /^(?:|I )fill in "(?P<field>(?:[^"]|\\")*)" with "(?P<value>(?:[^"]|\\")*)"$/
             | Fills in form field with specified id|name|label|value.
             | at `Behat\MinkExtension\Context\MinkContext::fillField()`

web_features | When /^(?:|I )fill in "(?P<field>(?:[^"]|\\")*)" with:$/
             | Fills in form field with specified id|name|label|value.
             | at `Behat\MinkExtension\Context\MinkContext::fillField()`

#...
```

That's it, you can now search and browse your whole step dictionary.

3.8 Configuration

3.8.1 Feature, Suite and Scenario Configuration

We already talked about configuring multiple contexts for a single test suite in a *previous chapter*. Now it is time to talk about test suites themselves. A test suite represents a group of concrete features together with the information on how to test them.

With suites you can configure Behat to test different kinds of features using different kinds of contexts and doing so in one run. Test suites are really powerful and `behat.yml` makes them that much more powerful:

```
# behat.yml

default:
    suites:
        core_features:
            paths:    [ %paths.base%/features/core ]
            contexts: [ CoreDomainContext ]
        user_features:
            paths:    [ %paths.base%/features/web ]
            filters:  { role: user }
            contexts: [ UserContext ]
        admin_features:
            paths:    [ %paths.base%/features/web ]
            filters:  { role: admin }
            contexts: [ AdminContext ]
```

Suite Paths

One of the most obvious settings of the suites is the `paths` configuration:

```
# behat.yml

default:
    suites:
        core_features:
            paths:
                - %paths.base%/features
                - %paths.base%/test/features
                - %paths.base%/vendor/.../features
```

As you might imagine, this option tells Behat where to search for test features. You could, for example, tell Behat to look into the `features/web` folder for features and test them with `WebContext`:

```
# behat.yml

default:
    suites:
        web_features:
            paths:    [ %paths.base%/features/web ]
            contexts: [ WebContext ]
```

You then might want to also describe some API-specific features in `features/api` and test them with an API-specific `ApiContext`. Easy:

```
# behat.yml
```

```
default:
    suites:
        web_features:
            paths:    [ %paths.base%/features/web ]
            contexts: [ WebContext ]
        api_features:
            paths:    [ %paths.base%/features/api ]
            contexts: [ ApiContext ]
```

This will cause Behat to:

1. Find all features inside `features/web` and test them using your `WebContext`.
2. Find all features inside `features/api` and test them using your `ApiContext`.

Note: `%paths.base%` is a special variable in `behat.yml` that refers to the folder in which `behat.yml` is stored.

Path-based suites are an easy way to test highly-modular applications where features are delivered by highly decoupled components. With suites you can test all of them together.

Suite Filters

In addition to being able to run features from different directories, we can run scenarios from the same directory, but filtered by specific criteria. The Gherkin parser comes bundled with a set of cool filters such as *tags* and *name* filters. You can use these filters to run features with specific tag (or name) in specific contexts:

```
# behat.yml

default:
    suites:
        web_features:
            paths:    [ %paths.base%/features ]
            contexts: [ WebContext ]
            filters:
                tags: @web
        api_features:
            paths:    [ %paths.base%/features ]
            contexts: [ ApiContext ]
            filters:
                tags: @api
```

This configuration will tell Behat to run features and scenarios tagged as `@web` in `WebContext` and features and scenarios tagged as `@api` in `ApiContext`. Even if they all are stored in the same folder. How cool is that? But it gets even better, because Gherkin 4+ (used in Behat 3+) added a very special *role* filter. That means, you can now have nice actor-based suites:

```
# behat.yml

default:
    suites:
        user_features:
            paths:    [ %paths.base%/features ]
            contexts: [ UserContext ]
            filters:
                role: user
        admin_features:
            paths:    [ %paths.base%/features ]
```

```
            contexts: [ AdminContext ]
            filters:
                role: admin
```

A Role filter takes a look into the feature description block:

```
Feature: Registering users
  In order to help more people use our system
  As an admin
  I need to be able to register more users
```

It looks for a `As a ...` or `As an ...` pattern and guesses its actor from it. It then filters features that do not have the expected actor from the set. In the case of our example, it basically means that features described from the perspective of the *user* actor will be tested in `UserContext` and features described from the perspective of the *admin* actor will be tested in `AdminContext`. Even if they are in the same folder.

While it is possible to specify filters as part of suite configuration, sometimes you will want to exclude certain scenarios across the suite, with the option to override the filters at the command line.

This is achieved by specifying the filter in the gherkin configuration:

```
# behat.yml

default:
    gherkin:
      filters:
        tags: ~@wip
```

In this instance, scenarios tagged as @wip will be ignored unless the CLI command is run with a custom filter, e.g.:

```
vendor/bin/behat --tags=wip
```

Tip: More details on identifying tests can be found in the chapter *Identifying Tests*.

Extensions

Extensions can be configured like this:

```
# behat.yml

default:
    extensions:
        Behat\MinkExtension:
            base_url: http://www.example.com
            selenium2: ~
```

Suite Contexts

Being able to specify a set of features with a set of contexts for these features inside the suite has a very interesting side-effect. You can specify the same features in two different suites being tested against different contexts *or* the same contexts configured differently. This basically means that you can use the same subset of features to develop different layers of your application with Behat:

```
# behat.yml

default:
```

```
    suites:
        domain_features:
            paths:    [ %paths.base%/features ]
            contexts: [ DomainContext ]
        web_features:
            paths:    [ %paths.base%/features ]
            contexts: [ WebContext ]
            filters:
                tags: @web
```

In this case, Behat will first run all the features from the `features/` folder in `DomainContext` and then only those tagged with `@web` in `WebContext`.

Tip: It might be worth reading how to *execute a specific suite* or *initializing a new suite*

3.8.2 Custom Autoloading

Sometimes you will need to place your `features` folder somewhere other than the default location (e.g. `app/features`). All you need to do is specify the path you want to autoload via `behat.yml`:

```
# behat.yml

default:
    autoload:
        '': %paths.base%/app/features/bootstrap
```

If you wish to namespace your features (for example: to be PSR-1 complaint) you will need to add the namespace to the classes and also tell behat where to load them. Here `contexts` is an array of classes:

```
# behat.yml

default:
    autoload:
        '': %paths.base%/app/features/bootstrap
    suites:
        default:
            contexts: [My\Application\Namespace\Bootstrap\FeatureContext]
```

Note: Using `behat.yml` to autoload will only allow for `PSR-0` You can also use `composer.json` to autoload, which will also allow for `PSR-4`

3.8.3 Formatters

Default formatters can be enabled by specifying them in the profile.

```
# behat.yml

default:
    formatters:
        pretty: true
```

CHAPTER 4

Cookbooks

4.1 Integrating Symfony2 with Behat

Symfony2 is a Web Application Framework that can be easily integrated and used seamlessly with Behat 3. As a prerequisite for this cookbook you need to have working Symfony2 application.

In this cookbook we will cover:

1. Installing Behat dependency with Composer.
2. Initialising Behat suite.
3. Installing and enabling Symfony2 extension.
4. Accessing application services in contexts.
5. Using Symfony2 test client as a Mink driver.

4.1.1 Installing Behat in your Symfony2 Project

Recommended way of managing Behat dependency in your project is to use Composer. Assuming that you already have `composer.json` file in your project you only need to add one new entry to it and install. It can be done automatically for you with this command:

```
$ php composer.phar require --dev behat/behat
```

Note: Note that we have used `--dev` switch for Composer. It means that Behat will be installed as as `require-dev` dependency in your project, and will not be present in production. For further information please check Composer documentation.

4.1.2 Initialising Behat

After execution of this command you should see information about files initialised in your project, and you should be able to write your first scenario. In order to verify Behat initialisation you can just run following command:

```
$ bin/behat
```

Tip: If you don't feel familiar with Behat enough please read *Quick Start* first.

4.1.3 Installing and Enabling Symfony2 Extension

Great, you have a Behat suite working in your project, now it's time to install Symfony2Extension. To do this you need to add another dependency, but in the same way we did it a while ago:

```
$ php composer.phar require --dev behat/symfony2-extension
```

Now it's time to enable extension in your `behat.yml` file. If it doesn't exist just create such file in your project root and fill it with following content:

```
default:
  extensions:
    Behat\Symfony2Extension: ~
```

If this file already exists just change its contents accordingly. From that point you should be able to run Behat and Symfony2 extension will be loaded and ready to work with.

4.1.4 Accessing Application Services in Contexts

The extension we have just installed detects the default Symfony configuration and allows to use your application services in context classes. To make a service available in a context you need to change your `behat.yml` configuration and tell the extension which services to inject:

```
default:
  suites:
    default:
        contexts:
            - FeatureContext:
                session:   '@session'
  extensions:
    Behat\Symfony2Extension: ~
```

This configuration will try to to match the `$session` dependency of your `FeatureContext` constructor by injecting the `session` service into the context. Be careful because if such a service does not exist or its name does not match, it will not work and you will end up with a Behat exception.

4.1.5 Using KernelDriver with your Behat Suite

Symfony2 has a build-in Test Client, which can help you with web acceptance testing, why not make use of it? Especially because Behat has a Mink Extension that makes those kind of testing even easier.

The advantage of using KernelDriver instead of standard Mink driver is that you don't need to run web server in order to access a page. Also you can even use Symfony Profiler and inspect your application directly!. You can read more about test client in Symfony Documentation.

If you don't have Mink and MinkExtension yet, you can install those two with:

```
$ php composer.phar require --dev behat/mink
$ php composer.phar require --dev behat/mink-extension
```

In order to install BrowserKit Driver you need to execute following command:

```
$ php composer.phar require --dev behat/mink-browserkit-driver
```

Now you are only one step from being ready to make full use of Symfony2 extension in your project. You need to enable extension in your `behat.yml` file as follows:

```
default:
  extensions:
    Behat\Symfony2Extension: ~
    Behat\MinkExtension:
      sessions:
        default:
          symfony2: ~
```

Et voilà! Now you are ready to drive your Symfony2 app development with Behat3!

4.2 Gathering Contexts when using Multiple Contexts

When splitting the definitions in multiple contexts, it might be useful to access a context from another one. This is particularly useful when migrating from Behat 2.x to replace subcontexts.

Behat allows to access the environment in *Hooks*, so other contexts can be retrieved using a `BeforeScenario` hook:

```
use Behat\Behat\Context\Context;
use Behat\Behat\Hook\Scope\BeforeScenarioScope;

class FeatureContext implements Context
{
    /** @var \Behat\MinkExtension\Context\MinkContext */
    private $minkContext;

    /** @BeforeScenario */
    public function gatherContexts(BeforeScenarioScope $scope)
    {
        $environment = $scope->getEnvironment();

        $this->minkContext = $environment->getContext('Behat\MinkExtension\Context\MinkContext');
    }
}
```

Caution: Circular references in context objects would prevent the PHP reference counting from collecting contexts at the end of each scenarios, forcing to wait for the garbage collector to run. This would increase the memory usage of your Behat run. To prevent that, it is better to avoid storing the environment itself in your context classes. It is also better to avoid creating circular references between different contexts.

CHAPTER 5

Useful Resources

5.1 Integrating Behat with PHPStorm

More information on integrating Behat with PHPStorm can be found in this blog post.

5.2 Behat cheat sheet

An interesting Behat and Mink cheat sheet developed by Jean-François Lépine