Chapters

writing features - gherkin language

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

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

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

gherkin syntax

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

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

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

  Scenario: A different situation
      ...
```

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

1. `Feature: Some terse yet descriptive text of what is desired` starts the feature and gives it a title. Learn more about features in the "Features" section.
2. Behat does not parse the next 3 lines of text. (In order to... As an... I want to...). These lines simply provide context to the people reading your feature, and describe the business value derived from the inclusion of the feature in your software.
3. `Scenario: Some determinable business situation` starts the scenario, and contains a description of the scenario. Learn more about scenarios in the "Scenarios" section.
4. The next 7 lines are the scenario steps, each of which is matched to a regular expression defined elsewhere. Learn more about steps in the "Steps" section.
5. `Scenario: A different situation` starts the next scenario, and so on.

When you're executing the feature, the trailing portion of each step (after keywords like `Given`, `And`, `When`, etc) is matched to a regular expression, which executes a

PHP callback function. You can read more about steps matching and execution in *Defining Reusable Actions - Step Definitions*.

features

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

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

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

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

In addition to basic scenarios, feature may contain scenario outlines and backgrounds.

scenarios

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

The following scenarios each have 3 steps:


```
Scenario: Wilson posts to his own blog
  Given I am logged in as Wilson
```

```
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."

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

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

scenario outlines

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

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

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

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

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

    Examples:
      | start | eat | left |
      |  12   |  5  |  7   |
      |  20   |  5  |  15  |
```

The Scenario outline steps provide a template which is never directly run. A Scenario Outline is run once for each row in the Examples section beneath it (not counting the first row of column headers).

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

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

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

> You can also use placeholders in Multiline Arguments.
>
> Your step definitions will never have to match the placeholder text itself, but rather the values replacing the placeholder.

So when running the first row of our example:

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

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

The scenario that is actually run is:

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

backgrounds

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

```
Feature: Multiple site support

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

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

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

steps

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

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

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

givens

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

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"
```

It's ok to call into the layer "inside" the UI layer here (in symfony: talk to the models).

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

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

whens

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

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

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

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


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

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

thens

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

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

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

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

and, but

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

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

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

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

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

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

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

multiline arguments

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

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

Multiline step arguments come in two flavours: tables or pystrings.

tables

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

```
Scenario:
  Given the following people exist:
    | name  | email           | phone |
    | Aslak | aslak@email.com | 123   |
```

```
    | Joe   | joe@email.com   | 234   |
    | Bryan | bryan@email.org | 456   |
```

Don't be confused with tables from scenario outlines - syntactically they are identical, but have a different purpose.

A matching definition for this step looks like this:

```php
/**
 * @Given /the following people exist:/
 */
public function thePeopleExist(TableNode $table)
{
    $hash = $table->getHash();
    foreach ($hash 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()`).

pystrings

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

```
Scenario:
  Given a blog post named "Random" with:
    """
    Some Title, Eh?
    ===============
    Here is the first paragraph of my blog post.
    Lorem ipsum dolor sit amet, consectetur adipiscing
    elit.
    """
```


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

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

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

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.

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.

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
```

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

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
```

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

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

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