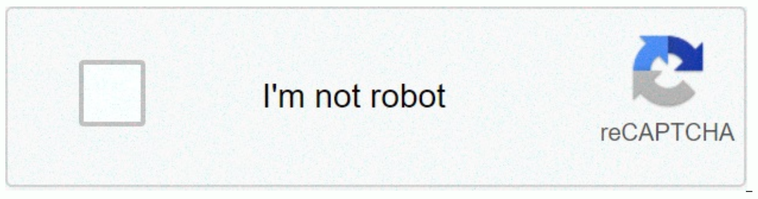

Continue

Acceptance criteria creation in gherkin format

Behat is a tool to test the behavior of your program, described in a special language called Gherkin. Gherkin is a business readable, domain-specific language created specifically to describe behavior. This gives you the ability to remove logical data from behavioral tests. The gherkin serves

two purposes: to serve as documentation of your project and automated tests. Behat also has a bonus feature: it talks to you using real, human language that tells you what code you should write. If you're still new to Behat, first jump into a quick introduction to Behata and then come back here to learn more about Gherkin. Like YAML or Python, the Gherkin is a line-oriented language that uses indentation to determine the structure. Line completions stop statements (called steps) and spaces or tabs can be used to indent. (We suggest that you use spaces for tolerance.) Finally, most of the lines in Gherkin begin with a special keyword: Feature: Some terse is still descriptive text of that what is desirable In order to realize the named business value As a clear system actor I want to get some profitable result, which is even more of a goal Scenario: Some defining business situations Given some prerequisite And some other prerequisites When some action of the actor and some other action And another action Then some test result is achieved and something else we can test going too script : another situation ... The parser divides input into functions, scripts, and steps. Let's look at the example above: Some terse is still descriptive text of what is desirable, runs the function and gives it a title. For more information about features, see Features. Behat does not analyze the next 3 lines of text. (In order to ... As... I want ...). These lines simply provide context to people reading your feature and describe the business value gained from incorporating a feature into your software. Scenario: Some a moderated business situation runs a script and contains a script description. Learn more about scenarios in the Scripts section. The next 7 rows are the script steps, each matching the regular expression defined elsewhere. Learn more about the steps in the Steps section. Scenario: Another situation begins with the following scenario and so on. When you perform this function, the end part of each step (after keywords such as Given, And, When, etc.) is mapped to a regular expression that performs the PHP callback function. You can read more about the steps that match and perform in determining reusable actions - defining steps. Each *.feature file conventionally consists of one function. Strings starting with keyword function: (or its localized equivalent), and then three lines with indentation triggers the function. The feature typically contains a list of scenarios. You can write whatever you want to the first script, which begins with a script: (or localized in the new line. Tags can be used to group functions and and regardless of the structure of the files and directories. Each script consists of a list of steps that should start with one of the keywords Given, When, Then, But or And (or localized). Behat treats them equally, but not worth it. Here's an example: Feature: Serve coffee In order to make money Customers should be able to buy coffee at all times Scenario: Buy the last coffee Given that there is 1 coffee left in the car And I deposited 1 dollar When I press the coffee button Then I should serve coffee In addition to the main scenarios, the function may contain contours of scripts and backgrounds. The script is one of Gherkin's main structures. Each script begins with a script: keyword (or localized), and then an optional script title. Each function can have one or more scenarios, and each script consists of one or both steps. The following scenarios each have 3 steps: Script: Wilson posts in his own blog Given that I logged in as Wilson When I try to post in Expensive Therapy then I should see your article was published. Script: Wilson fails to post to someone else's blog Given that I logged in as Wilson When I try to post in Greg's Anti-Tax Rants Then I Should See Gay! This is not your blog! Scenario: Greg posts on client blog Given that I logged in as Greg When I try to post in Road Therapy then I should see your article has been published. Copying and pasting scripts to use different values can quickly become boring and repetitive: Scenario: Eat 5 of 12 considering that there are 12 cucumbers When I eat 5 cucumbers Then I should have 7 cucumbers Scenario: Eat 5 out of 20 Given that there are 20 cucumbers When I eat 5 cucumbers Then I have to have 15 cucumbers Scenario Contours allow us to more concisely express these examples using a pattern with placeholders : Scenario Outline: Food Considering there are cucumbers When I eat cucumbers Then I should have <start> <eat> <left>cucumbers Examples: | start | eat | Left | | 12 | 5 | 7 | | 20 | 5 | 15 | Script structure steps provide a template that never directly runs. The script structure runs once for each row in the Examples section below it (not counting the first row of column headings). The script structure uses placeholders that <> contained in the script structure steps. For example: Considering <I'm a= placeholder= and= i'm= ok=>think of the placeholder as a variable. It will be replaced with a real value from the Examples table row: , where the text between the corner brackets of the placeholder matches the column header of the table. The value replaced by the placeholder changes with each subsequent run of the script structure until the end of the Examples table is reached. Tip You can also use placeholders in multiline arguments. Note Definition should never match the placeholder text itself, but rather the values replacing the placeholder. So, when you run the first line of our example: Script structure: control</I'm> </left> </eat> </start> </start> Considering there are <start>cucumbers When I eat <eat>cucumbers Then I have to <left>have cucumbers Examples: | Start | eat | Left | | 12 | 5 | 7 | Script that actually works: Scenario Outline: Order Control # replaced <start>by 12: Given that there are 12 cucumbers # <eat>replaced by 5: When I eat 5 cucumbers # <left>replaced by 7: Then I have to have 7 cucumbers Fony allows you to add some context to all scenarios in one function. The background is similar to an untitled script that contains several steps. The difference is when it runs: the background runs in front of each of your scripts, but after your hooks BeforeScenario (Connect to test process - Hooks). Feature: Multiple support site Background: Given a global admin called Greg I blog called Greg Antipodean Rants And a client named Wilson I blog called Expensive Therapy owned by Wilson Script: Wilson posts in his own blog Given I logged in as Wilson When I try to publish Expensive Therapy Then I should see your article has been published. Scenario: Greg posts on client blog Given that I logged in as Greg When I try to post in Road Therapy then I should see your article has been published. Features consist of steps, also known as Givens, when it is and then. Behat technically does not distinguish between these three types of steps. However, we strongly recommend that you do this! These words have been carefully selected for their purpose and you need to know that the goal is to get into the thinking of BDD. Robert S. Martin wrote a great post about the concept of BDD Given-When-Then, where he thinks of them as a finishing state apparatus. The purpose of this step is to put the system in a known state before the user (or external system) starts interacting with the system (in when steps). Avoid talking about user interactions nowadays. If you have worked with usage cases, your prerequisites are given. Note Two good examples of using Givens are: To create records (instances of a model) or to configure a database: Given that there are no users on the site Given that the database is pure authenticate user (exception to the recommendation without interaction. things that happened earlier in order): Given that I logged in as Everzet Tip it is ok to call in a layer inside the interface layer here (in symfony: talk to models). And for all symfony users out there, we recommend using this step with table arguments to configure entries instead of fixtures. So you can read the script all in one place and make sense of it without having to jump between files: Given that there are users: | username | password | e-mail | | Everzette Hotel | 123456 | everzet@knplabs.com | Fabba | 22@222 | fabpot@symfony.com | The goal when steps are to describe the key actions performed by the user (or by using Robert S. Martin's metaphor, перехід). Примітка Два гарні приклади використання: взаємодія з веб-сторінкою (бібліотека норок дає вам багато веб-дружніх коли кроки</left> </eat> </start> </left> </eat> </start> </start> box): When I am on / some / page When I fill in the username with everzet When I fill in the password with 123456 When I click login Interact with some CLI library (call commands and output recording): Aim then steps to observe the results. Observations should be related to business cost/benefit in describing your function. Observations should check the log output (report, user interface, message, exit command) rather than something deeply buried inside it (which has no business value and is instead part of implementation). Make sure that something related to Given +When there is (or not) in output Check if some external systems have received the expected message (was an email with certain content successfully sent?) when I call echo hello Then the output should be hi Note Although it may be tempting to take then steps to just look in the database - resist temptation. You should only check the output that is observed by the user (or external system). The database data itself is displayed only inside the program, but then is then permanently exposed to your system output in a web browser, at a command prompt, or an e-mail message. If you have multiple given, When or then the steps you can write: Script: A few given Dana one thing Dana another thing Dana still 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 that allows your script to read more freely: Script: A few given dana one thing And one more thing, and another thing when I open my eyes Then I see something, but I don't see something else if you prefer. you can step back the script steps in a more programmatic way, in many ways in the same way your actual code is receding to provide visual context: Script: Multiple given dana one thing And one more thing when I open my eyes then I see something But I don't see something else Behat interprets steps starting with and or but exactly like all the other steps. It's no different between them – you have to! A regular expression that matches steps allows you to capture small lines from your steps and receive them in the step definition. However, sometimes you need to transfer a richer data structure from step to step definition. This is what multiline step arguments are for. They are written in rows immediately after the step and are passed to the step definition method as the last argument. Multiline step-by-step arguments come in two flavors: tables or. Tables as arguments to steps are handy for defining a larger data set - usually as input to a given one or, as expected, output from then on. Scenario: Given the following people exist: | name | e-mail | Phone | | Aslak Hotel | aslak@email.com | 123 | | Joe | joe@email.com | 234 | | Brian | bryan@email.org | 456 | Note Do not confuse with tables from contour scripts - syntactically they are identical, but have a different purpose. The corresponding definition for this step is : /** * @Given /there are the following people:/ */ public function thePeopleExist(TableNode $table) { $hash = $table->getHash(); foreach ($hash as $row) { // $row['name'], $row['email'], $row['phone'] } } Note The table is entered into the definition as a TableNode object from which you can retrieve the hash by columns (TableNode Method::getHash() or rows (TableNode::getRowsHash()). The text must be compensated by delimiters consisting of three double quotation marks () on the lines by themselves: Script: Given a blog post called Random of: Some Titles, Yeah? ========== Here is the first paragraph of my blog. Note The inspiration for PyString comes from Python, where it is used to demarcate docstrings, largely in the way /* ... */ is used for multiline docblocks in PHP. Tip There is no need to find this text in the step definition and map it to your regular expression. The text will be automatically passed as the last argument in the step definition method. For example: /** * @Given /a blog called ([^]+) with:/ */ public blogPost($title, PyStringNode $markdown) { $this->createPost($title, $markdown->getRaw()); } PyStrings notes are stored in an instance of PyStringNode that you can simply convert to a string from (string) $pystring or $pystring->getRaw() as in the example above. Note Retreat discovery is not important, although common practice is two spaces with a fencing step. The retreat inside triple quotes, however, is significant. Each row of the string passed to the reverse call definition step will be defused according to the original value. This saves the indent outside the opening column. The 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 whether Behat and Gherkin support your language (e.g. French), run: behat --story-syntax --lang=fr Note Keep in mind that any language other than en must be explicitly marked with # : ... comment at the beginning of the file *.feature: # language: fr Fonctionnalité: ... Therefore, your features will store all information about its content type, which is very important for methodologies such as BDD, and will also give Behat the ability to have multi-language features in one package. Suite.