FREE eBook

LEARNING behat

Free unaffiliated eBook created from
Stack Overflow contributors.

#behat

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: behat

It is an unofficial and free behat ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official behat.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with behat

Remarks

This section provides an overview of what behat is, and why a developer might want to use it.

It should also mention any large subjects within behat, and link out to the related topics. Since the Documentation for behat is new, you may need to create initial versions of those related topics.

Examples

Functional testing as user stories

Functional tests are best described as tests for your user stories. If you've dealt with user stories before they normally follow the following pattern:

```
As a [role], I want to [desire], so that [benefit/desired outcome]
```

For the following examples we'll use this user story as an example:

```
As a Dungeon Master, I want to ensure harmony and mutual trust, so that
the party can work together as a team
```

The two most popular testing frameworks for functional tests in PHP are Behat and PHPSpec.

Beginning with Behat

Behat provides Gherkin Syntax which is a human-readable format. It allows you to easily describe your user stories.

To start with Behat you should install it with Composer and then initialize your test files:

```
$ composer require --dev behat/behat="^3.0.5"

$ ./vendor/bin/behat --init

+d features # place your *.feature files here
+d features/bootstrap # place your context classes here
+f features/bootstrap/FeatureContext.php # place your definitions, transformations and hooks
here
```

By default you place your test files in the `features/` folder and have the extension `.feature`.

Each test file should define a particular feature of the application. A feature is broken down into a bunch of scenarios and contain a series of steps that need to be performed successfully for the scenario to pass. Each scenario needs to pass for a feature to pass.

```
# features/PartyHarmony.feature
Feature: Party Harmony
    As a Dungeon Master, I want to ensure harmony and mutual trust, so that
    the party can work together as a team

    Scenario: Teach members to respect each others property
        Given that the Wizard has 10 cookies
        And the Bard eats 1 cookie
        Then the Bard is mysteriously on fire
```

To run your tests, you execute the Behat binary directly. We can optionally specify which feature file to run (otherwise all tests are run). This feature file will fail with undefined steps errors (because we haven't defined what those steps mean):

```
$ ./vendor/bin/behat features/PartyHarmony.feature
Feature: Party Harmony
    As a Dungeon Master, I want to ensure harmony and mutual trust, so that
    the party can work together as a team

  Scenario: Teach members to respect each others property # features/PartyHarmony.feature:6
    Given that the Wizard has 10 cookies
    And the Bard eats 1 cookie
    Then the Bard is mysteriously on fire

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.01s (10.49Mb)

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

    /**
     * @Given that the Wizard has :arg1 cookies
     */
    public function thatTheWizardHasCookies($arg1)
    {
        throw new PendingException();
    }

    /**
     * @Given the Bard eats :arg1 cookie
     */
    public function theBardEatsCookie($arg1)
    {
        throw new PendingException();
    }

    /**
     * @Then the Bard is mysteriously on fire
     */
    public function theBardIsMysteriouslyOnFire()
    {
        throw new PendingException();
    }
```

Each step in a scenario runs a piece of code from a context PHP file (different feature tests can load different contexts). We can copy the examples suggested by Behat or create our own ones. The step are matched to a regular expression check. So if we implement

```
<?php
#
class FeatureContext {
    /**
     * @Given that the wizard has :num cookies
     */
    public function wizardHasCookies($num) {
        // $this->wizard is a pre-existing condition.... like syphilis
        $this->wizard->setNumberOfCookies($num);
    }

    /**
     * @Given the Bard eats :num cookie
     */
    public function theBardEatsCookie($num)
    {
        $this->bard->consumeCookies($num);
    }

    /**
     * @Then the Bard is mysteriously on fire
     */
    public function theBardIsMysteriouslyOnFire() {
        PHPUnit_Framework_Assert::assertTrue(
            $this->bard->isBardOnFire()
        );
    }
}
```

You'll notice the use of `PHPUnit_Framework_Assert`. Behat doesn't have it's own assert system so you can use whichever one you want.

Now running the tests will execute actual code and we can test if everything passes:

```
$ ./vendor/bin/behat features/PartyHarmony.feature
Feature: Party Harmony
    As a Dungeon Master, I want to ensure harmony and mutual trust, so that
    the party can work together as a team

  Scenario: Teach members to respect each others property # features/PartyHarmony.feature:6
    Given that the Wizard has 10 cookies                  #
FeatureContext::thatTheWizardHasCookies()
    And the Bard eats 1 cookie                            #
FeatureContext::theBardEatsCookie()
    Then the Bard is mysteriously on fire                 #
FeatureContext::theBardIsMysteriouslyOnFire()

1 scenario (1 passed)
3 steps (3 passed)
0m0.01s (10.59Mb)
```

Extending Behat with Mink

Mink provides an interface for web drivers (like Goutte and Selenium) as well as a `MinkContext` which, when extended, provides additional web language for our steps.

To install Mink (and the default Goutte driver):

```
$ composer require --dev behat/mink-extension="^2.0"
$ composer require --dev behat/mink-goutte-driver="^1.0"
```

Then extend your context with `MinkContext`:

```
<?php
use Behat\MinkExtension\Context\MinkContext;

class FeatureContext extends MinkContext … {
    …
}
```

You can see the entire list of syntax available in your Behat install with the following command:

```
$ ./vendor/bin/behat -dl

Given /^(?:|I )am on "(?P<page>[^"]+)"$/
 When /^(?:|I )reload the page$/
 When /^(?:|I )move backward one page$/
 When /^(?:|I )move forward one page$/
 When /^(?:|I )press "(?P<button>(?:[^"]|\\")*)"$/
 When /^(?:|I )follow "(?P<link>(?:[^"]|\\")*)"$/
 When /^(?:|I )fill in "(?P<field>(?:[^"]|\\")*)" with "(?P<value>(?:[^"]|\\")*)"$/
```

You then need to configure Mink to indicate where the website you want to test is located and which Web Drivers to use (Goutte by default):

```
# ./behat.yml
default:
    extensions:
        Behat\MinkExtension:
            base_url: "[your website URL]"
            sessions:
                default:
                    goutte: ~
```

Here is an example of a scenario using only the Mink provided steps:

```
# ./features/Authentication.feature
Feature: Authentication
    As a security conscious developer I wish to ensure that only valid users can access our
website.

    Scenario: Login in successfully to my website
        When I am on "/login"
        And I fill in "email" with "my@email.com"
        And I fill in "password" with "my_password"
        And I press "Login"
        Then I should see "Successfully logged in"

    Scenario: Attempt to login with invalid credentials
        When I am on "/login"
        And I fill in "email" with "my@email.com"
        And I fill in "password" with "not_my_password"
        And I press "Login"
        Then I should see "Login failed"
```

You can now test this by running the feature via Behat:

```
./vendor/bin/behat features/Authentication.feature
```

You can create your own steps using `MinkContext` for common steps (for instance logging in is a very common operation):

```
Feature: Authentication
    As a security conscious developer I wish to ensure that only valid users can access our
website.

    Scenario: Login in successfully to my website
        Given I login as "my@email.com" with password "my_password"
        Then I should see "Successfully logged in"

    Scenario: Attempt to login with invalid credentials
        Given I login as "my@email.com" with password "not_my_password"
        Then I should see "Login failed"
```

You will need to extend your context file with the `MinkContext` to get access to the web drivers and page interactions:

```
<?php
use Behat\MinkExtension\Context\MinkContext;

class FeatureContext extends MinkContext {
    /**
      * @Given I login as :username with password :password
      */
    public function iLoginAsWithPassword($username, $password) {
        $this->visit("/login");
        $this->fillField("email", $username);
        $this->fillField("password", $password);
        $this->pressButton("Login");
    }
}
```

Mink also provides CSS selectors in most of it's pre-provided calls which allows you to identify elements on the page using constructs such as this:

```
When I click on "div[id^='some-name']"
And I click on ".some-class:first"
And I click on "//html/body/table/thead/tr/th[first()]"
```

Testing JavaScript with Mink and Selenium

If we want to test JavaScript on a website, we'll need to use something a bit more powerful than Goutte (which is just cURL via Guzzle). There are a couple of options such as ZombieJS, Selenium and Sahi. For this example I'll use Selenium.

First you'll need to install the drivers for Mink:

```
$ composer require --dev behat/mink-selenium2-driver="^1.2"
```

And you'll also need to download the Selenium standalone server jar file and start it:

```
$ java -jar selenium-server-standalone-2.*.jar
```

We'll also need to tell Behat that when we use the `@javascript` tag to use the Selenium driver and provide the Selenium standalone server's location.

```
# ./behat.yml
default:
    # …
    extensions:
        Behat\MinkExtension:
        base_url: "[your website URL]"
        sessions:
            # …
            javascript:
                selenium2:
                    browser: "firefox"
                    wd_host: http://localhost:4444/wd/hub
```

Then, for each test you want to be run using browser emulation, you just need to add a `@javascript` (or `@selenium2`) tag to the beginning of the feature or scenario.

```
# ./features/TestSomeJavascriptThing.feature
@javascript # or we could use @selenium2
Feature: This test will be run with browser emulation
```

The test can then be run via Behat (as any other test). The one difference is, when the test is run it should spawn a browser window on the computer running the Selenium standalone server which will then perform the tests described.

Setting up test data

With functional testing data is often modified. This can cause subsequent runs of the test suite to fail (as the data could have changed from the original state it was in).

If you have setup your data source using an ORM or framework that supports migration or seeding (like Doctrine, Propel, Laravel), you can use this to create a new test database complete with Fixture data on each test run.

If you don't currently use one of these (or equivalent), you can use tools like Phinx to quickly setup a new test database or prepare an existing database for each test run (clean up test entries, reset data back to it's original state).

```
# Install Phinx in your project
$ php composer.phar require robmorgan/phinx

$ php vendor/bin/phinx init
Phinx by Rob Morgan - https://phinx.org. version x.x.x
```

```
Created ./phinx.xml
```

Add your database credentials to `./phinx.xml`.

```
$ php vendor/bin/phinx create InitialMigration
```

You can specify how your database tables are created and populated using the syntax provided in the documentation.

Then, each time you run your tests you run a script like this:

```
#!/usr/bin/env bash

# Define the test database you'll use
DATABASE="test-database"

# Clean up and re-create this database and its contents
mysql -e "DROP DATABASE IF EXISTS $DATABASE"
mysql -e "CREATE DATABASE $DATABASE"
vendor/bin/phinx migrate

# Start your application using the test database (passed as an environment variable)
# You can access the value with $_ENV['database']
database=$DATABASE php -d variables_order=EGPCS -S localhost:8080

# Run your functional tests
vendor/bin/behat
```

Now your functional tests should not fail due to data changes.

Capturing emails

Functional testing also can include testing processes that leave your environment, such as external API calls and emails.

As an example, imagine you're functionally testing the registration process for your website. The final step of this process involves sending an email with an activation link. Until this link is visited, the account isn't completely registered. You'd want to test both:

1. That the email will be send correctly (formatting, placeholder replacement, etc) and,
2. That the activation link works

Now you could test the email sending but using an IMAP or POP client to retrieve the sent email from the mailbox, but this means you're also testing your Internet connection, the remote email server and any problems that may arise in delivery (spam detection for instance).

A simpler solution is to use a local service that traps outgoing SMTP connections and dumps the sent email to disk.

A couple of examples are:

smtp-sink - A utility program that comes bundled with Postfix.

```
# Stop the currently running service
sudo service postfix stop

# Dumps outgoing emails to file as "day.hour.minute.second"
smtp-sink -d "%d.%H.%M.%S" localhost:2500 1000

# Now send mails to your local SMTP server as normal and they will be
# dumped to raw text file for you to open and read

# Run your functional tests
vendor/bin/behat
```

Don't forget to kill `smtp-sink` and restart your postfix service afterwards:

```
# Restart postfix
sudo service postfix start
```

FakeSMTP - A Java-based client that traps outgoing mail

```
# -b = Start without GUI interface
# -o = Which directory to dump your emails to
$ java -jar fakeSMTP.jar -b -o output_directory_name
```

Alternatively you can use a remote service that provides this service like mailtrap but then your testing is dependent on Internet access.

Installation or Setup

Behat/Mink

Install using composer (for other methods check) behat.org If you using linux, please go sure that you have installed php-curl (normal curl installation won't work)

Linux

```
sudo apt-get install php5-curl
```

If you are using **Windows**, make sure you have PHP, Curl and Git installed. You can find those under following links:

- PHP (Xampp) : https://www.apachefriends.org/de/index.html
- Curl: http://curl.haxx.se/latest.cgi?curl=win64-nossl
- Git: http://git-scm.com/download/win

Your composer.json would contain the following:

behat - composer.json

```
{
  "require": {
    "behat/behat": "dev-master",
```

```
    "behat/mink": "dev-master",
    "behat/mink-extension": "dev-master",
    "behat/mink-selenium2-driver": "dev-master",
    "phpunit/php-code-coverage": "dev-master",
    "phpunit/phpunit-mock-objects": "dev-master",
    "phpunit/phpunit": "dev-master"
  },
  "minimum-stability": "dev",
  "config": {
    "bin-dir": "bin/"
  }
}
```

(when saving the composer.json file in Windows, you need to choose "All files" as Filetype and "ANSI" coding)

Afterwards execute the following commands:

```
$ curl http://getcomposer.org/installer | php
$ php composer.phar install
```

After this Behat, Mink and Behat-Mink extension are installed, To execute behat

execute behat

```
$ bin/behat
```

To activate the Behat-Mink Extension use: behat.yml create a file "behat.yml" with the following content

behat.yml

```
default:
  suites:
    default:
      paths:
        features: %paths.base%/features/
        bootstrap: %paths.base%/features/bootstrap/
      contexts:
        - FeatureContext
  extensions:
    Behat\MinkExtension:
      base_url: 'http://www.startTestUrl.de'
      selenium2:
        browser: firefox
        wd_host: "http://localhost:4444/wd/hub"
```

This file will be in the same directory that contains bin directory and link to behat.
Also note that in the yml file, do not use tabs for indentation. use spaces. To get a list of commands available in behat-mink, use

```
$ bin/behat -di
```

Make behat part of your system

Linux

Go to your Homedirectory and do the following:

```
$ sudo vi .bashrc
```

And add this lines at the end of the directory

```
export BEHAT_HOME=/home/*user*/path/to/behat
export PATH=$BEHAT_HOME/bin:$PATH
```

Restart the console or type "source .bashrc"

Windows

Go over the Systemsettings and add the Path of behat/bin to the environment-variables

Other Drivers Over drivers like Selenium, phantomjs, goutte, etc. must be installed too.

Read Getting started with behat online: https://riptutorial.com/behat/topic/9136/getting-started-with-behat

Credits

S. No	Chapters	Contributors
1	Getting started with behat	Community, dasmelch, Tom