Behat Page Object Extension Documentation

Release 2.0

Jakub Zalas

March 16, 2016

Contents

1 Guide **3**
- 1.1 Installation . . . 3
- 1.2 Introduction to page objects . . . 3
- 1.3 Working with page objects . . . 4
- 1.4 Working with page object elements . . . 8
- 1.5 Writing assertions . . . 10
- 1.6 Configuration . . . 11

2 Cookbooks **13**
- 2.1 Creating a custom factory . . . 13

This Behat extension provides tools for implementing page object pattern.

Page object pattern is a way of keeping your context files clean by separating UI knowledge from the actions and assertions. Read more on the page object pattern on the Selenium wiki.

CHAPTER 1

Guide

1.1 Installation

This extension requires:

- Behat 3.0+

1.1.1 Through Composer

The easiest way to keep your suite updated is to use Composer.

1. Require the extension in your `composer.json`:

```
$ composer require --dev sensiolabs/behat-page-object-extension:^2.0
```

2. Activate the extension in your `behat.yml`:

```
default:
  # ...
  extensions:
    SensioLabs\Behat\PageObjectExtension: ~
```

1.2 Introduction to page objects

Page object encapsulates all the dirty details of a user interface. Instead of messing with the page internals in our context files, we'd rather ask a page object to do this for us:

```
<?php

/**
 * @Given /^(?:|I )change my password$/
 */
public function iChangeMyPassword()
{
    // $page = get page...
    $page->login('kuba', '123123')
       ->changePassword('abcabc')
       ->logout();
}
```

Page objects hide the UI and expose clean services (like login or changePassword), which can be used in the context classes. On one side, page objects are facing the developer by providing him a clean interface to interact with the pages. On the other side, they're facing the HTML, being the only thing that has knowledge about a structure of a page.

The idea is we end up with much cleaner context classes and avoid duplication. Since page objects group similar concepts together, they are easier to maintain. For example, instead of having a concept of a login form in multiple contexts, we'd only store it in one page object.

1.3 Working with page objects

1.3.1 Creating a page object class

To create a new page object extend the `SensioLabs\Behat\PageObjectExtension\PageObject\Page` class:

```php
<?php

namespace Page;

use SensioLabs\Behat\PageObjectExtension\PageObject\Page;

class Homepage extends Page
{
}
```

1.3.2 Instantiating a page object

Injecting page objects into a context file

Page objects will be injected directly into a context file if they're defined as constructor arguments with a type hint:

```php
<?php

use Behat\Behat\Context\Context;
use Page\Homepage;
use Page\Element\Navigation;

class SearchContext implements Context
{
    private $homepage;
    private $navigation;

    public function __construct(Homepage $homepage, Navigation $navigation)
    {
        $this->homepage = $homepage;
        $this->navigation = $navigation;
    }

    /**
     * @Given /^(?:|I )visited homepage$/
     */
    public function iVisitedThePage()
    {
```

```
            $this->homepage->open();
        }
    }
```

Using the page object factory

Note: `PageObjectContext` described here is deprecated in favour of injecting page objects directly into context's constructor (described in the previous section). `PageObjectContext` will be removed in future versions of this extension.

Pages are created with a built in factory. One of the ways to use them in your context is to call the `getPage` method provided by the `SensioLabs\Behat\PageObjectExtension\Context\PageObjectContext`:

```
    <?php

    use SensioLabs\Behat\PageObjectExtension\Context\PageObjectContext;

    class SearchContext extends PageObjectContext
    {
        /**
         * @Given /^(?:|I )search for (?P<keywords>.*?)$/
         */
        public function iSearchFor($keywords)
        {
            $this->getPage('Homepage')->search($keywords);
        }
    }
```

Note: Alternatively you could implement the `SensioLabs\Behat\PageObjectExtension\Context\PageObjectA`

Page factory finds a corresponding class by the passed name:

- *"Homepage"* becomes a *"Homepage"* class
- *"Article list"* becomes an *"ArticleList"* class
- *"My awesome page"* becomes a *"MyAwesomePage"* class

Note: It is possible to implement your own way of mapping a page name to an appropriate page object with a custom factory.

1.3.3 Opening a page

Page can be opened by calling the `open()` method:

```
    <?php

    use Behat\Behat\Context\Context;
    use SensioLabs\Behat\PageObjectExtension\Context\PageObjectContext;

    class SearchContext implements Context
```

```
{
    private $homepage;
    private $navigation;

    public function __construct(Homepage $homepage, Navigation $navigation)
    {
        $this->homepage = $homepage;
        $this->navigation = $navigation;
    }

    /**
     * @Given /^(?:|I )visited (?:|the )(?P<pageName>.*?)$/
     */
    public function iVisitedThePage($pageName)
    {
        if (!isset($this->$pageName)) {
            throw new \RuntimeException(sprintf('Unrecognised page: "%s".', $pageName));
        }

        $this->$pageName->open();
    }
}
```

However, to be able to do this we have to provide a `$path` property:

```
<?php

namespace Page;

use SensioLabs\Behat\PageObjectExtension\PageObject\Page;

class Homepage extends Page
{
    /**
     * @var string $path
     */
    protected $path = '/';
}
```

Note: `$path` represents an URL of your page. You can omit the `$path` if your page object is only returned from other pages and you're not planning on opening it directly. `$path` is only used if you call `open()` on the page.

Path can also be parametrised:

```
protected $path = '/employees/{employeeId}/messages';
```

Any parameters should be given to the `open()` method:

```
$this->getPage($pageName)->open(array('employeeId' => 13));
```

It's also possible to check if a given page is opened with `isOpen()` method:

```
$isOpen = $this->getPage($pageName)->isOpen(array('employeeId' => 13));
```

Both `open()` and `isOpen()` run the same verifications, which can be overriden:

- `verifyResponse()` - verifies if the response was successful. It only works for drivers which support getting a response status code.
- `verifyUrl()` - verifies if the current URL matches the expected one. The default implementation only checks if a page url is exactly the same as the current url. Override this method to implement your custom matching logic. The method should throw an exception in case URLs don't match.
- `verifyPage()` - verifies if the page content matches the expected content. It is up to you to implement the logic here. The method should throw an exception in case the content expected to be present on the page is not there.

1.3.4 Implementing page objects

Page is an instance of a Mink DocumentElement. This means that instead of accessing `Mink` or `Session` objects, we can take advantage of existing Mink Element methods:

```php
<?php

namespace Page;

use Behat\Mink\Exception\ElementNotFoundException;
use SensioLabs\Behat\PageObjectExtension\PageObject\Page;

class Homepage extends Page
{
    // ...

    /**
     * @param string $keywords
     *
     * @return Page
     */
    public function search($keywords)
    {
        $searchForm = $this->find('css', 'form#search');

        if (!$searchForm) {
            throw new ElementNotFoundException($this->getDriver(), 'form', 'css', 'form#search')
        }

        $searchForm->fillField('q', $keywords);
        $searchForm->pressButton('Google Search');

        return $this->getPage('Search results');
    }
}
```

Notice that after clicking the *Search* button we'll be redirected to a search results page. Our method reflects this intent and returns another page by creating it with a `getPage()` helper method first. Pages are created with the same factory which is used in the context files.

Reference the official Mink API documentation for a full list of available methods:

- DocumentElement
- TraversableElement
- Element

Note that when using page objects, the context files are only responsible for calling methods on the page objects and making assertions. It's important to make this separation and avoid assertions in the page objects in general.

Page objects should either return other page objects or provide ways to access attributes of a page (like a title).

1.4 Working with page object elements

Page object doesn't have to relate to a whole page. It could also correspond to some part of it - an element. Elements are page objects representing a section of a page.

1.4.1 Inline elements

The simplest way to use elements is to define them inline in the page class:

```php
<?php

namespace Page;

use SensioLabs\Behat\PageObjectExtension\PageObject\Page;

class Homepage extends Page
{
    // ...

    protected $elements = array(
        'Search form' => 'form#search',
        'Navigation' => array('css' => '.header div.navigation'),
        'Article list' => array('xpath' => '//*[contains(@class, "content")]//ul[contains(@class
    );

    /**
     * @param string $keywords
     *
     * @return Page
     */
    public function search($keywords)
    {
        $searchForm = $this->getElement('Search form');
        $searchForm->fillField('q', $keywords);
        $searchForm->pressButton('Google Search');

        return $this->getPage('Search results');
    }
}
```

The advantage of this approach is that all the important page elements are defined in one place and we can reference them from multiple methods.

The *$elements* array should be a list of selectors indexed by element names. The selector can be either a string or an array. If it's a string, a css selector is assumed. The key of an array is used otherwise.

The difference between the *getElement()* method and Mink's *find()*, is that the later might return *null*, while the first will throw and exception when an element is not found on the page.

1.4.2 Custom elements

In case of a very complex page, the page class might grow too big and become hard to maintain. In such scenarios one option is to extract part of the logic into a dedicated element class.

To create an element we need to extend the `SensioLabs\Behat\PageObjectExtension\PageObject\Element` class. Here's a previous search example modeled as an element:

```php
<?php

namespace Page\Element;

use SensioLabs\Behat\PageObjectExtension\PageObject\Element;
use SensioLabs\Behat\PageObjectExtension\PageObject\Page;

class SearchForm extends Element
{
    /**
     * @var array|string $selector
     */
    protected $selector = '.content form#search';

    /**
     * @param string $keywords
     *
     * @return Page
     */
    public function search($keywords)
    {
        $this->fillField('q', $keywords);
        $this->pressButton('Google Search');

        return $this->getPage('Search results');
    }
}
```

Defining the `$selector` property is optional but recommended. When defined, it will limit all the operations on the page to the area within the selector. Any selector supported by Mink can be used here.

Similarly to the inline elements, the selector can be either a string or an array. If it's a string, a css selector is assumed. The key of an array is used otherwise.

Accessing custom elements is much like accessing inline ones:

```php
<?php

namespace Page;

use SensioLabs\Behat\PageObjectExtension\PageObject\Page;

class Homepage extends Page
{
    // ...

    /**
     * @param string $keywords
     *
     * @return Page
     */
    public function search($keywords)
```

```
        {
            return $this->getElement('Search form')->search($keywords);
        }
    }
```

Note: Page factory takes care of creating custom elements and their class names follow the same rules as Page class names.

Element is an instance of a NodeElement, so similarly to pages, we can take advantage of existing Mink Element methods. Main difference is we have more methods relating to the single `NodeElement`. Reference the official Mink API documentation for a full list of available methods:

- NodeElement
- TraversableElement
- Element

1.5 Writing assertions

Page objects are our interface to the web pages. We still need context files though, not only to call the page objects, but also to verify expectations.

Traditionally we'd want to throw exceptions if expectations are not met. The difference is we'd ask a page object to provide needed page details instead of retrieving them ourselves in the context file:

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

    class ConferenceContext implements Context
    {
        private $conferenceList;

        public function __construct(ConferenceList $conferenceList)
        {
            $this->conferenceList = $conferenceList;
        }

        /**
         * @Then /^(?:|I )should not be able to enrol to (?:|the )"(?P<conferenceName>[^"]*)" confer
         */
        public function iShouldNotBeAbleToEnrolToTheConference($conferenceName)
        {
            if ($this->conferenceList->hasEnrolmentButtonFor($conferenceName)) {
                $message = sprintf('Did not expect to find an enrollment button for the "%s" confere

                throw new \LogicException($message);
            }
        }
    }
```

Our page object could look like the following:

```
    namespace Page;

    class ConferenceList extends Page
    {
```

```
        public function hasEnrolmentButtonFor($conferenceName)
        {
            $conferenceSlug = str_replace(' ', '-', strtolower($conferenceName));
            $button = $this->find('css', sprintf('#enrol-%s', $conferenceSlug));

            return !is_null($button);
        }
    }
```

We could go one step fruther in making our life easier by using phpspec matchers available through the expect() helper:

```
    /**
     * @Then /^(?:|I )should not be able to enrol to (?:|the )"(?P<conferenceName>[^"]*)" conference
     */
    public function iShouldNotBeAbleToEnrolToTheConference($conferenceName)
    {
        expect($this->getPage('Conference list'))->notToHaveEnrolmentButtonFor($conferenceName);
    }
```

To use the expect() helper, we need to install it first. Best way to do this is by adding it to the `composer.json`:

```
    "require-dev": {
        "bossa/phpspec2-expect": "~1.0"
    }
```

1.6 Configuration

If you use namespaces with Behat, we'll try to guess the location of your page objects. The convention is to store pages in the `Page` directory located in the same place where your context files are. Elements should go into additional `Element` subdirectory.

Defaults can be simply changed in the `behat.yml` file:

```
    default:
      extensions:
        SensioLabs\Behat\PageObjectExtension:
          namespaces:
            page: [Acme\Features\Context\Page, Acme\Page]
            element: [Acme\Features\Context\Page\Element, Acme\Page\Element]
          factory:
            id: acme.page_object.factory
            page_parameters:
              base_url: http://localhost
```

CHAPTER 2

Cookbooks

2.1 Creating a custom factory

To implement a custom page object factory the `SensioLabs\Behat\PageObjectExtension\PageObject\Factory` needs to be implemented, and registered as a service within your extension.

Id of the service has to be then configured in the `behat.yml`:

```
default:
  extensions:
    SensioLabs\Behat\PageObjectExtension:
      factory: acme.page_object.factory
```

2.1.1 Custom class name resolver with the default factory

In most cases the default page object factory should meet our needs, and what we'd like to overload is the class name resolver.

Class name resolver is used by the default factory to actually convert the page object name to a class namespace.

To implement a custom class name resolver the `SensioLabs\Behat\PageObjectExtension\PageObject\Factory\Cla` needs to be implemented and registered as a service within your extension.

Id of the service has to be then configured in the `behat.yml`:

```
default:
  extensions:
    SensioLabs\Behat\PageObjectExtension:
      factory:
        id: ~ #optional
        class_name_resolver: acme.page_object.class_name_resolver
```