www.phparch.com

May 2021
Volume 20 - Issue 5

php[architect]

Testing Assumptions

On the Road to Long Living PHP

Let's Discover Atoum

Streamlining Behat

Free Sample Article

ALSO INSIDE

PHP Puzzles:
Sending and Receiving Polybius Ciphers

Education Station:
How to Build a REST API

The Workshop:
New OS, Old PHP

Community Corner:
Interview with Ryan Weaver

Sustainable PHP:
Wizard Thinking

Security Corner:
Radical Transparency

finally:
Awkward Conversations

Streamlining Behat

Oscar Merida

Behat, and feature tests overall, are crucial for giving you confidence that when you refactor code, you don't change how an application works and for helping you understand how an application behaves in practice. Investing in integration and functional tests is worth it in the long term, and in this article, we'll see how I also updated my Behat setup to run browser tests in Chrome.

On a recent client project, we're modernizing and fixing a WordPress-based application with a ton of custom code, business logic, and design. As is usual in these cases, we were brought in to sort out a legacy codebase that's been through multiple developer's hands where the focus was on getting things working. Unfortunately, technical debt, and code quality, were afterthoughts at best, and we inherited multiple similarly named files with identically named functions with curiously similar code at a cursory glance.

Early on, I asked the development lead if I could use Behat to write tests. I wanted to get the infrastructure in place for the long term, and I pitched writing a few to ensure the homepage loaded and then to walk through the purchase path on the site. Given the project was in a startup-like mode of "get things working" driven by business needs to bring in revenue, I expected some push back. Instead, I was delighted and surprised when he readily agreed and gave me the go-ahead.

Listing 1.

```
{
    "require": {
        "behat/behat": "^3.3",
        "behat/mink": "^1.7",
        "behat/mink-extension": "^2.2",
        "behat/mink-selenium2-driver": "^1.3",
        "behat/mink-goutte-driver": "^1.2"
    }
}
```

Listing 2.

```
default:
  suites:
    default:
      contexts:
        - FeatureContext:
            arg:
              environment: dev
  extensions:
    Behat\MinkExtension:
      goutte: ~
      javascript_session: selenium2
      selenium2:
        wd_host: http://127.0.0.1:4444/wd/hub
      base_url: http://local.example.com
```

Old Behat Configs

One thing I realized is that my usual setup for Behat was a bit dated. I'd had some issues getting it to work originally, so I tended to leave it alone once it was in a working state. Part of the complication was using Selenium[1] to control browsers. While the promise of being able to use any supported browser was enticing, in practice, I'd never needed to do so.

My `composer.json` file wasn't too complicated, but I have to confess that I arrived at it from trial-and-error and wasn't sure if I needed all the files specified as in Listing 1.

On the other hand, `behat.yaml` I'd essentially built by copying-and-pasting working code from elsewhere. See Listing 2. I did not feel confident that any changes to it wouldn't break my setup. In practice, upgrading selenium or Firefox, at the time, usually risked something breaking in an API somewhere. Which, you guessed it, meant I avoided upgrading things. It's the circle of life.

Updating Behat

I took advantage of a nearly-blank slate to put in more sound foundations. Instead of using Selenium, which I'd picked since I wanted to run tests in Firefox, I switched to using Chrome directly via chromedriver. Firefox wasn't an option when I looked into it because its equivalent API to chromedriver wasn't feature-complete. This may not be the case anymore, see "GeckoDriver vs. Marionette: Differences"[2], and it is worth re-evaluating.

Still, chucking Selenium slims down the number of moving parts. So instead of talking to Chrome via Selenium, we speak to it directly—one less thing to run.

1 *Selenium: https://www.selenium.dev*

2 *"GeckoDriver vs. Marionette: Differences": http://phpa.me/gecko-vs-marionette*

```
{
    "require": {
        "behat/behat": "^3.8",
        "dmore/behat-chrome-extension": "^1.3",
        "friends-of-behat/mink-extension": "^2.5",
        "behat/mink-browserkit-driver": "^1.3"
    }
}
```

One thing that we discussed early on in this engagement was where to put the tests. I prefer to treat my test suite as its own project. That means it gets its own `composer.json` *directory in a* `tests/` *folder away from other code. Since these aren't unit tests and interact with the website via a browser client, this setup delineates the separation. Testing code is nowhere near the project's web root. The code that gets deployed with each release and its dependencies don't impact the main project's dependencies.*

I took the time to understand `behat.yaml` better. For one, I discovered the `autoload` key that tells Behat's PSR-0 autoloader to also look in our local `bootstrap` directory to load classes. The `suites` section lets us configure the `Contexts` available in our tests. We'll come back to it later. The two lines under `extensions` set up the `ChromeExtension` driver to talk to Chrome and tell the `MinkExtension` how to talk to it via the `api_url` setting. See Listing 3.

We have a working setup that runs through all the tests in the `features/` folder from our `tests/` directory. Since most of the tests require chrome to be running, we wrote a script around the commands needed to execute all the tests. The entire script has more features, mainly to run on non-Linux machines, but essentially it executes the commands shown in Listing 4.

Simplifying the commands to run through the tests was essential to ensuring the team would run them before submitting a pull request.

Listing 3.

```
default:
  autoload:
    '': "%paths.base%/bootstrap"
  suites:
    default:
      contexts:
        - FeatureContext:
  extensions:
    DMore\ChromeExtension\Behat\ServiceContainer\ChromeExtension: ~
    Behat\MinkExtension:
      browser_name: chrome
      sessions:
        default:
          chrome:
            api_url: "http://127.0.0.1:9222"
            socket_timeout: 60
```

Listing 4.

```
CHROME_EXE=`which google-chrome`

# start chrome so we can stop it when done
"$CHROME_EXE" --remote-debugging-address=0.0.0.0 --remote-debugging-port=9222 \
      --ignore-certificate-errors --disable-gpu --headless > /dev/null 2>&1 &
CHROME_PID=$!

# give chrome some time to launch
sleep 1

./vendor/bin/behat --format=progress ./features

# stop chrome
kill $CHROME_PID;
```

Organizing Features

Behat calls the code that provides the rules for what you're testing "Features" and provides a default `boostrap/FeatureContext.php` file where you can put them. It's a class holding the PHP code that executes when the matching scenario step is called.

It doesn't take much before that file is brimming with tests, many unrelated to them. Instead of letting it grow uncontrollably, take some time to move related ones into independent contexts. On this site, we quickly had two clear functionalities. First, users can log into the site, and they can book trips through it. Each becomes a Context in `behat.yaml` like so:

```
suites:
  default:
    contexts:
      - FeatureContext:
      - BookingContext:
      - LoginContext:
```

Reusing Inputs

Frequently, you'll need to provide inputs to a step. Instead of hardcoding those values in a scenario, I like to put those in a `yaml` file loaded into each feature. A context can take one or more arguments defined in `behat.yaml`:

```
- LoginContext:
   settingsYAML: "%paths.base%/shared.yml"
```

The constructor for the feature can parse that data as in Listing 5.

Listing 5.

```php
<?php

use Behat\MinkExtension\Context\RawMinkContext;
use Symfony\Component\Yaml\Yaml;

class LoginContext extends RawMinkContext
{
    private $settings;

    /**
     * Initializes context.
     */
    public function __construct(string $settingsYAML) {
        if (empty($this->settings)) {
            $this->settings = Yaml::parse(
                file_get_contents($settingsYAML)
            );
        }
    }
```

Steps can use the values in `$this->setttings`:

```php
/**
 * @When I submit the login details for :arg1 User
 */
public function iSubmitTheLoginDetailsFor($creds) {

    if (!isset($this->settings['Users'][$creds])) {
        $this->raiseError("Can't find user $creds");
    }
```

And, finally, this keeps the login scenarios readable—more on that in a moment. Furthermore, we can reuse these same values across scenarios. Because they're defined in one place, we Don't Repeat Ourselves, so we stay DRY.

```
Scenario: A registered user can log in and see the user dashboard
    When I go to "/dashboard/"
    Then I should be on the "/login/" page
    And I submit the login details for "Testing" User
    Then I should see the welcome message for "Testing" User
    And I should see the referral code for "Testing" User
```

Finally, our steps are more reusable since they can work with any set of values, like user credentials that conform to the expected structure.

```
Scenario: Show usesr an error message if the password is wrong
    When I go to "/login/"
    And I submit the login details for "TestingBadPassword" User
```

Call for Speakers is Open!

Share your knowledge with PHP developers in beautiful Austin, Texas on October 14-16, 2021!

Whether you're in town, across the country, or across an ocean, submit your presentation by June 30th for consideration.

We're looking forward to helping a few hundred developers level up their craft and carry forward the PHP community torch in-person in October.

View Speaker Package
https://cfp.longhornphp.com

LONGHORN
PHP CONFERENCE

Writing Scenarios

My first experiences with Behat were for testing Drupal sites with the Drupal Extension[3]. Unfortunately, one of the habits it encourages for writing Gherkin tests is to mix knowledge of page structure into the tests. This leads to unreadable tests when you're looking for various elements. Consider the following scenario. Now imagine we want to verify that dozens of HTML elements are visible. Should our Gherkin test know about the themes in our Drupal region? Could non-technical stakeholders read such a test and make sense of it with 12 or more elements, CSS selectors, and region names? Those look like implementation details to me.

```
Scenario: Homepage Contact Us Link
  Given I am on the homepage
  Then I should see the link "Contact Us" in the "branding_second"
 region
  Then I should see the "Search" button in the "branding_second"
 region
  Then I should see the "div#block-system-main-menu" element in
the "menu" region
```

Nowadays, I prefer to write tests that hide HTML and CSS details from the reader. They cut directly to what should be on the page without cluttering it with where or how. That detail is for writing custom steps.

```
@javascript
Scenario: An anonymous user can start to book a tour
    Given I go to a bookable tour page
    And I see the pricing table
    Then I click on the book now button for the bookable date
```

Unleashing CSS Selectors

Well, how do we interact with the web pages we're testing in PHP? It's not a browser. However, the Mink project—remember seeing something about Mink above?—provides an API for us to inspect and interact with. First, look back at the last scenario above. The `@javascript` tag above the starting line of each tells Behat we're going to test a page with, well, JavaScript interactivity. You can omit it for basic tests, and they'll run faster. However, most applications depend heavily on it, so the number of useful scenarios that don't use it is small.

Let's look at a step from the "book a tour" test shown earlier.

```
And I see the pricing table
```

Our `BookingContext` defines a method. Behat uses a comment starting with `@Then` to match that method to the readable English step we used there. The `@Then` is a placeholder. In Gherkin, it lets us write "Then I see the pricing table" or "And I see the Pricing table" and a few other alternatives to keep a nice flow to our steps. After we get the current `$page` from the browser session, we use the `find()` method to look for an element using a `'css'` selector. In this case, we're looking for something, presumably a table, with an ID of `dates-pricing-table`. If it's not found, we throw an `\Exception` to indicate the test failed as in Listing 6.

Listing 6.

```php
/**
 * @Then I see the pricing table
 */
public function iSeePricingTable()
{
   $page = $this->getSession()->getPage();
   $table = $page->find('css', '#dates-pricing table');

   if (empty($table)) {
      throw \Exception("Pricing table not found");
   }
}
```

Since Features are classes, your free to add helper methods to your tests when you identify similar tasks. I have one called `findByCSS()` that condenses getting the page and issuing a query into one step. Behat's CSS selectors are also powerful. Almost anything you can do in JavaScript to locate a DOM node will work in step definition. For example, the snippet in Listing 7 looks for an error message when invalid login credentials are entered in the login form. We can search for an `input` element with a specific `name` attribute. If we find it, we can get its parent node and look within it for children with a `.validation` class.

Listing 7.

```php
<?php
/**
 * @Then I see an error message for wrong login credentials
 */
public function iSeeAnErrorMessageForWrongLoginCredentials() {

   $form = $this->findByCSS('#login_form');
   $emailInput = $form->find('css', 'input[name=email]');

  $error = $emailInput->getParent()->find('css', '.validation');

   if (!$error) {
      throw new \Exception("Can't find error message.");
   }

   if ($error->getText() !== self::ERROR_BAD_CREDENTIALS) {
      throw new \Excpetion(
         "Wrong error message for wrong login credentials."
      );
   }
}
```

3 *Drupal Extension: http://phpa.me/drupal-extension*

Behat's CSS selectors and its more specialized "named" selectors[4] should cover you when you're looking for an HTML element. The following step uses the `named` selector to look for a button with either that `name` attribute or the matching text button. If it's found, we can call `click()` to interact with it and move to the next step in our scenario. See Listing 8.

Listing 8.

```php
/**
 * @Then I click on the :btnNameOrText button
 */
public function iClickOnTheButton($btnNameOrText)
{
   $page = $this->getSession()->getPage();
   $btn = $this->page->find('named', ['button', $btnNameOrText]);
   if (empty($btn)) {
      throw new \Exception("Button '$btnNameOrText' not found");
   }

   $btn->click();
}
```

Spinning

How do you account for interactivity on a web page? By default, behat assumes you are navigating between pages and handles navigating a site well. But what if the user clicks a button that uses JavaScript to refresh part of the DOM? A naive solution is to pepper your tests or steps with `sleep()` calls and hope you pick values long enough for your tests to pass but not so long that the test suite takes forever to run. Oh, and they should also work across developer machines and other testing environments. It's honestly a no-win approach. Instead, Behat supports "spin" functions. These are callables that look for an element on the page, up to a specified timeout.

This part of a test waits for an image slider to load before checking its contents are correct (Listing 9).

Listing 9.

```php
/**
 * @Then I can click to see :slide in the itinerary slide
 */
public function iCanClickToSeeInTheItinerarySlide($slide)
{
   // wait for the slider to load and initialze
   $this->spin(
      function ($context) {
         // expect to return false or throw an exception if
         // we're waiting for an element to load
         $node = $context->findByCSS(
            "#itinerarySlider1Nav a.slider-nav-item"
         );

         return !empty($node->getText());
      },
      15 // seconds
   );

   // rest of testing code continues
```

These are called "spin" functions because the built-in `spin()` method will execute it at intervals. It'll stop if the callable returns `true`, in this case, because we've found an itinerary slider item and continue executing. If it reaches the 15-second timeout, an `\Exception` is thrown to halt the test.

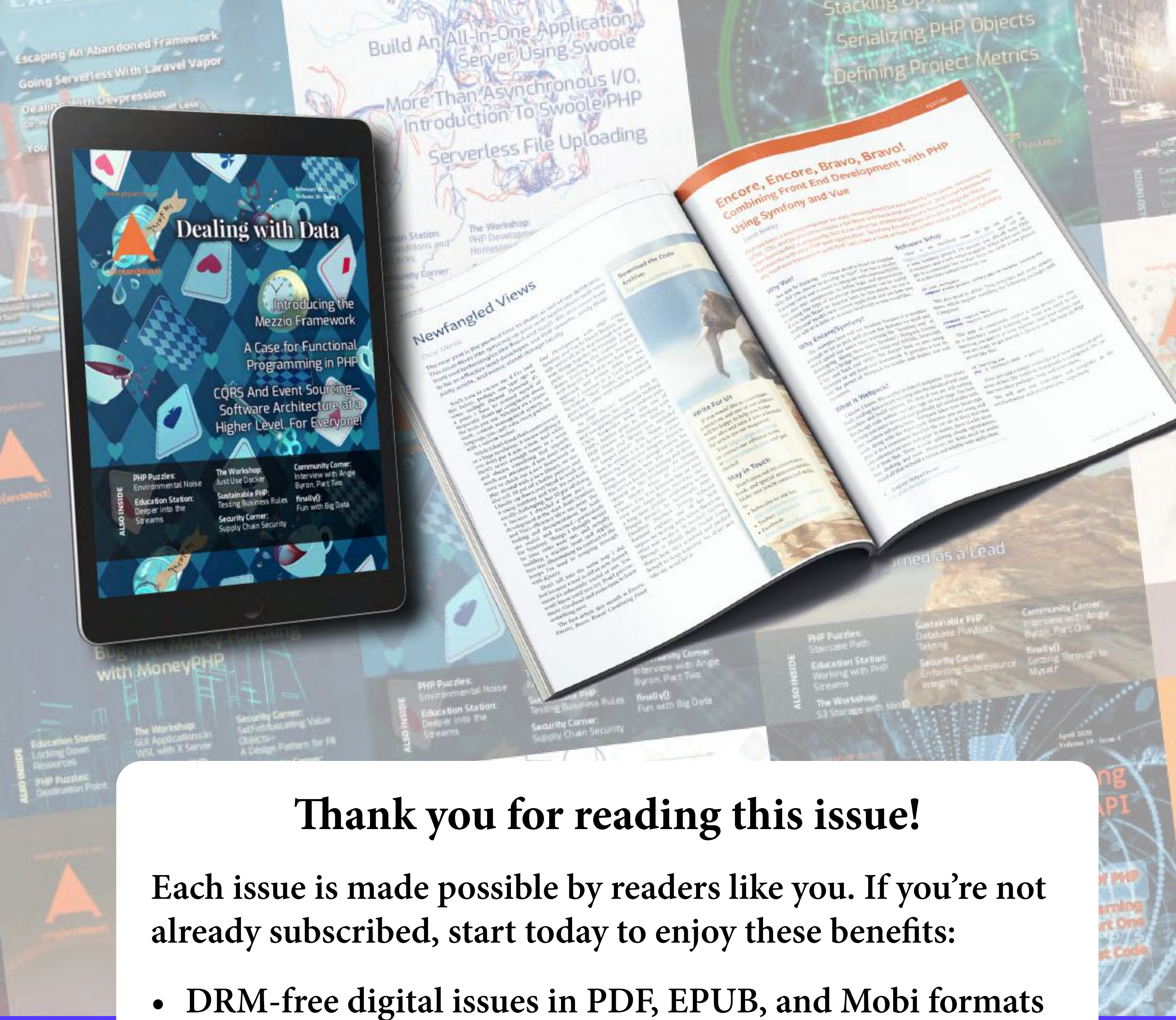
Conclusion

Taking the time to clean up cruft from my "it works" testing skeleton was worth it. The cleaner setup has fewer dependencies. Second, I learned to write tests that don't scare away non-coders, are more maintainable. Finally, spin functions and CSS selectors are helpful in writing feature tests that work on a modern, JavaScript-heavy UI.

Related Reading

- *The Workshop: Specification BDD with Phpspec* by Joe Ferguson, May 2020. https://phpa.me/workshop-may-20
- *Capturing an API's Behavior With Behat* by Michael Heap, January 2017. https://www.phparch.com/magazine/2017-2/january/
- *Leveling Up: Phpspec, TDD, and Mock Objects* by David Stockton, May 2015. https://www.phparch.com/magazine/2015-2/may/

4 *"named" selectors: http://phpa.me/behat-css-expression*


Dealing with Data
Introducing the Mezzio Framework
A Case for Functional Programming in PHP
CQRS And Event Sourcing–Software Architecture at a Higher Level, For Everyone!
Encore, Encore, Bravo, Bravo! Combining Front End Development with PHP Using Symfony and Vue
Newfangled Views
Build An All-In-One Application Server Using Swoole
More Than Asynchronous I/O, Introduction To Swoole PHP
Serverless File Uploading
Serializing PHP Objects
Defining Project Metrics