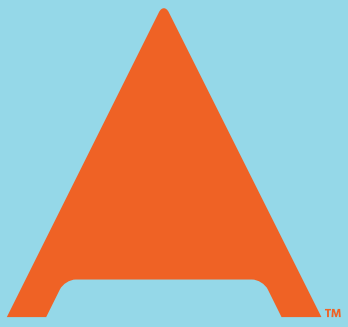
www.phparch.com

December 2016
VOLUME 15 - Issue 12



php[architect]

Scrutinizing Your Tests

FREE Article!

Behat: Beyond Browser Automation

Strangler Pattern, Part Three: the Rhythm of Test-Driven Development

ALSO INSIDE

Decoupled Blocks With Drupal 8 and JavaScript Frameworks

Abstracting HTTP Clients in PHP

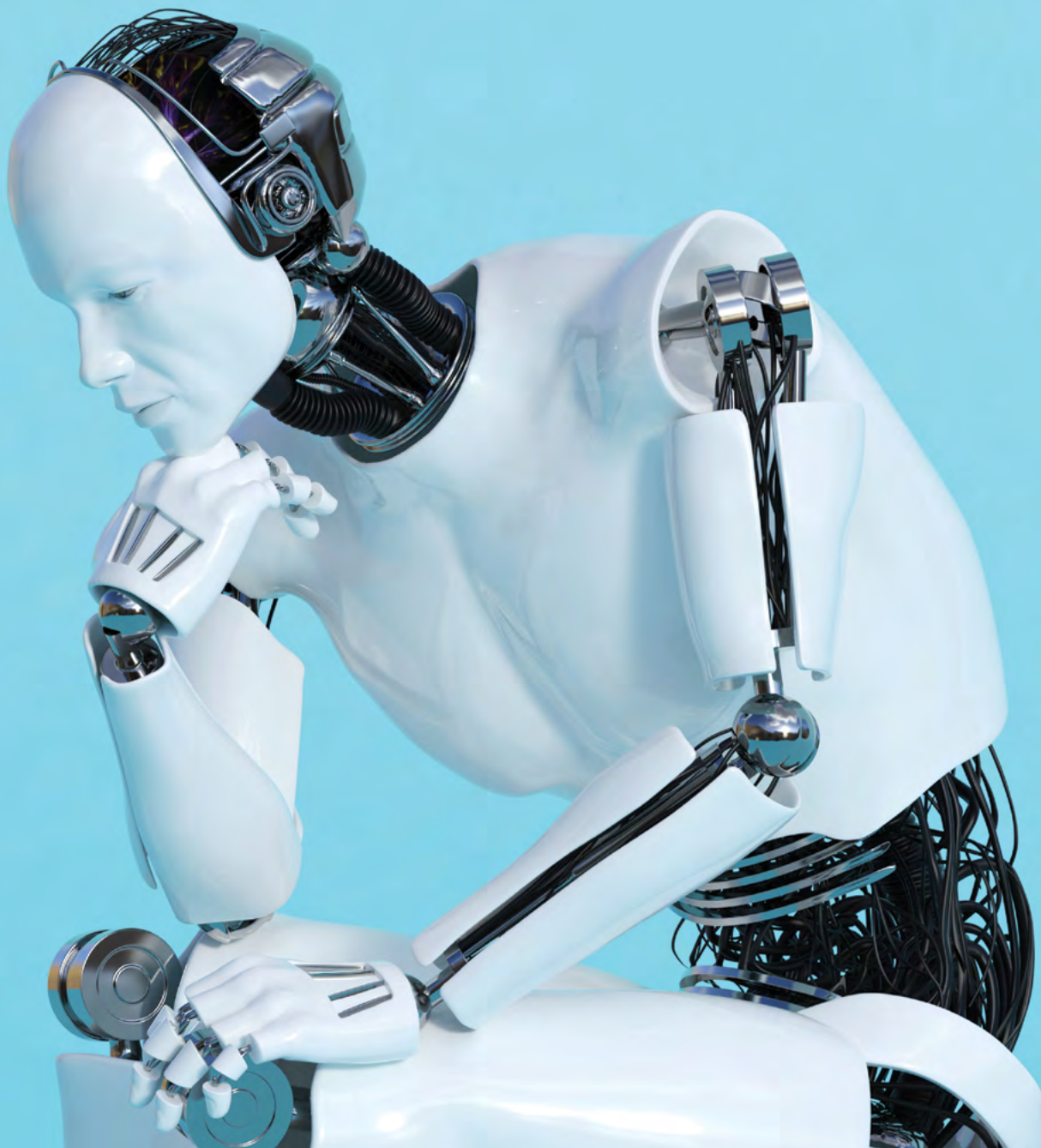
Education Station: Let's Build a Chatbot in PHP

Community Corner: Focus on What We Have in Common

Leveling Up: Building Better Bug Reports

Security Corner: Keeping a Secret

finally{}: The Year of ???



Behat: Beyond Browser Automation

Konstantin Kudryashov

Behat is a tool written in PHP to support teams in practicing Behavior-driven Development. In its simplest form, it is a test automation tool which focuses on comprehensibility more than it does on validity. Behat provides you with a simple developer-agnostic language—Gherkin, and then gives you automation capabilities on top of it. Gherkin's semantic flexibility is both its biggest asset and its biggest flaw.

The First Years

In the Gherkin[1], and by extension Behat[2] world, this:

```
Scenario: Product costing less than 10 results in \
delivery cost of 3
  Given there is a product with SKU "RS1" and a cost of \
5 in the catalog
  And I am on "/catalog"
  When I press "Add RS1 to the basket"
  And I follow "My Basket"
  Then I should see "Total price: 9"
```

is as good of a feature as this:

```
Scenario: Product costing less than 10 results in \
delivery cost of 3
  Given there is a product with SKU "RS1" and a cost \
of 5 in the catalog
  When I add the product with SKU "RS1" from the \
catalog to my basket
  Then the total price of my basket should be 9
```

However, when you consider the larger implications of the evolution of these features, the second one is leaps and bounds better than the first. We will come back to the exact reasoning later in this article. As with any "evolution," the difference is very hard to spot on the first sight, before you get to more complicated cases.

When I first started developing Behat, its ability to test applications end-to-end through the browser was the most attractive feature. It was about time I tried to get into Test-driven Development, so the simplicity of interface-focused tests became the answer to that drive and a little obsession of mine. This drive was so entrenched in my thinking in the first Behat years you couldn't hear me talking about Behat without talking about a user interface. Perhaps the greatest example of this overwhelming focus on UI automation was how quick after initial introduction of Behat I created and integrated it with Mink[3]—the browser automation tool. Back then, browser automation and Behat were almost indistinguishable concepts for me. Deep Mink integration, default Mink step definitions, and the rest were the product of that time. But things have changed drastically since then; I learned how wrong I was about that approach.

I remember when I first heard from my Ruby friend about this cool new testing tool—Cucumber[4]. Cucumber and its integration with Capybara[5] seemed like an incredible story of taking something as complex as TDD and making it as simple as writing down interactions your users should have with the website. I started looking for a possibility to test PHP applications with Cucumber. There were ways, but they all shared a similar drawback—even if your tests are end-to-end, you do need to have the ability to quickly and easily access the application persistence to clean the environment, or do a test setup. Obviously, this was a fairly tricky proposition for a testing tool written in Ruby. I was essentially forced to start writing a PHP implementation of Cucumber—something destined to become Behat.

I love to tell this story because it highlights how our methods and tools shape our thinking. As I stated previously, my first years with Behat went under the sign of browser automation. As the time went on, though, things started to change rapidly. I like to think the first year in Behat life was all about me developing Behat and the rest of our years together were all about Behat developing me. Through development of the tool I learned the way I was using and understanding the tool were far from optimal. That's when I stepped on the transitional path from being a Behat user to being a Behavior-driven Development practitioner. And with every new step on this path, browser automation was losing its hold on me.

1 *Gherkin:* https://cucumber.io/docs/reference
2 *Behat:* http://behat.org
3 *Mink:* http://mink.behat.org
4 *Cucumber:* https://cucumber.io
5 *Capybara:* https://github.com/jnicklas/capybara

Behavior-Driven Development

BDD[6] is a methodology coined by Dan North and Chris Matts. It is a collaborative process created as an outcome of collaboration between a developer (Dan North) and a business analyst (Chris Matts). BDD was developed as a way to bring practice as technical as TDD to the context as non-technical as traditional business analysis. Dan and Chris recognized there's not much difference between how our code and our business processes go. From their perspective, every computer program and every business process follow the same flow:

1. When the action is performed
2. Then, the output is produced

Every method call, every function has the same reason to its existence— when called (action is performed), it produces some effect (output is produced) like saving an entity to the database, checking form validity, or switching DNS records. Interestingly, every business interaction or a process has the same effect—every tax form submission, every support phone call has the same purpose—producing outputs (or outcomes). The only addition Chris made based on his experience is that he introduced the context into the mix:

1. Given the context
2. When the action is performed
3. Then, the output is produced

6 *BDD:* http://behaviourdriven.org

Context was the third and the last important part of this equation. From this point onwards BDD became a way to have conversations about systems at different levels of said systems, both business and technical ones. When given a task of implementing a feature (e.g. "product delivery cost") you would ask, "can you give me an example of this?" and from this point you would channel the discussion into the Given-When-Then structure. The structure here is the interesting part, not the keywords themselves (Given-When-Then). Sometimes, your examples might lack any of above keywords yet would still follow the Context-Action-Output structure. Structure is an interesting bit, not the syntax.

Another great thing about BDD is how devoid it is from implementation or technical details, which is intentional. By disconnecting our problem discussion from our solution finding processes Dan and Chris effectively created an explicit space for teams to explore the problem domain enough without being overwhelmed by one single solution. Database layer, architecture, programming language used, or even the UI being employed, all these are left outside of said discussion. That is the lost essence of Behavior-driven Development.

Gherkin and Browser Automation

When you create something as seminal as BDD, people notice and start following. Technical people tend to follow with processes and tools. Eventually, Given-When-Then spawned the language-agnostic specification format called Gherkin, and a little later the tool employing the format for test automation—Cucumber. Aslak, the creator of Cucumber, famously said Cucumber is the world's most misunderstood automation tool (because of the lack of interest in underlying principles). As you probably guessed from the beginning of this article, I can proudly say I was one of the first people who completely misunderstood the underlying principles of Cucumber.

Gherkin made it very simple to provide an automation layer for your application without much knowledge of programming languages, design, testing, or even development practices. This fact alone attracted thousands of people to both Cucumber and Behat as test automation tools devoid of a need to worry much about good practices of design or automation. Easiness of testing brought by these tools suddenly meant thousands of people who otherwise didn't know where to start with automation now had a very clear path—installing a tool and writing a "BDD test" in Gherkin. Sadly, as with anything "very simple" and "trivial," crucial drawbacks are hidden in using Behat and Cucumber this way.

I worked on tens of projects employing both BDD as a practice and Behat as an automation tool and many of them shared the same problem— something which seemed like a good idea at the beginning of a project ended up being a huge block towards the middle of it. One such great idea which turned bad was using your Gherkin features primarily for

browser automation. It is easy to see why using Gherkin for browser automation might seem like a great idea at the beginning; no architecture implications, no test setup, or technical skills required. But why does this turn out to be a bad idea in the long run? In my experience there are two reasons why browser automation kills your test suite and BDD practice in general:

1. User interface tests are notoriously slow and brittle.
2. Scenarios focused on user interface are tightly coupled to a particular implementation.

In the following sections, I'll address each of these points individually and try to explain why are they such a big problem for your usage of Behat.

User Interface Tests Are Notoriously Slow and Brittle

Your test is as stable as the least stable layer of the stack it is going through. Your test is as fast as the slowest layer in the stack it is going through. Let's say we are describing the delivery cost calculation for a particular a basket. Let's also assume we are using extensively rich UI driven by AJAX and the discount logic is a combination of core objects and services. In this case, each of our end-to-end Behat tests will execute in the following manner:

1. Behat executes a single step in the scenario.
2. The step sends a command to Selenium server.
3. Selenium server sends a command to the JavaScript block injected into the browser.
4. JavaScript block injected into the browser forces an operation inside the browser window.
5. Browser performs and operation forcing it to send an AJAX request to the server.
6. Server receives a request and parses it into a controller call.
7. Controller delegates the call to the combination of core objects and services.
8. Core objects and services do calculation and return the result back to the controller.
9. Controller packs the result into an AJAX response and sends it back to the browser.
10. Browser receives the response and updates the UI.
11. UI update forces JavaScript block injected into the browser to notify Selenium server.
12. Selenium server notifies back the step in your Scenario.
13. Behat goes to the second step in your Scenario.

This is what happens every time you call `followLink()` in your step definition, sometimes tens of times per step definition. In real world terms, this is somewhere close to a 1s of your test execution time. The interesting part begins when you start asking the question: out of this entire stack, what is the single most important thing for the "delivery cost calculation" test you wrote? It's step eight: "Core objects and services do calculation and return the result back to the controller." Everything else is an indirection necessary to render the interface and capture the output from the user securely. User input/output must be tested, but should it be tested together with the "bundled discount price for all the products in a basket" logic? That's quite a question.

Effectiveness of tests as with anything else in the world is defined by the costs and values they add to the process. What is the cost of a test? It's the time it takes to write and maintain a test! What is the value of a test? It is two-fold; part of it is about helping you design the right solution correctly and another part is about protecting the right solution from degradation or breakage. Let's look closely at how our 13-layers delivery cost test performs in terms of its cost and value.

First, UI tests are notoriously cheap to implement the first time around, but what I am interested in is how cheap are they to maintain. How often do you think your business will ask you to change the delivery cost calculations? My guess would be every time business rethinks its business model, operations and marketing strategy, in other words—not very often. How often do you think you will need to update the UI? Based on my experience, every time a new major JS framework is released. Likely, every couple of minutes. Jokes aside, the rate of UI change is obviously faster than the rate of change in the business logic, by a huge margin! What do you think happens when each element of your business logic is tested through UI? You are artificially linking rates of change for both, which results in the smallest common denominator— every time your UX team moves the basket controls around the UI you are forced to fix all your tests, even the ones testing the calculations which didn't change a single bit. That results in a test suite which is extremely costly to maintain, because you are effectively forced to update the majority of your test suite every time UI is updated. And, UI is frequently updated. Otherwise, your business will end up being far behind its competition.

OK, entangling UI and core business logic inside your tests drastically increases the cost of maintaining your test suite, so what? Surely the value these end-to-end tests provide us outweigh the cost. Except they don't! Remember, the value your tests are supposed to deliver is all about design support and reduction of breakages. So how do your UI-focused tests help with the core object design? They don't, because all the business logic is now hidden behind levels of indirection such as UI and persistence. Even more, all the information your Gherkin scenarios are obsessed with is how a particular button is called or where is it located on the page, not how the discount calculation is supposed to work.

How about a reduction in breakages? Ask yourself, when was the last time your UI test highlighted that you broke the logic rather than the fact you broke the test by changing the CSS class of a button. Every single time your test fails without the underlying logic (the one being actually tested) being

broken, you pay a high price; you and your team loose trust in your test suite, one bit at a time. Your test suite becomes less and less reliable every minute. Maintaining the test suite stops being a useful activity to support the team's progress and starts being a chore.

I've observed many teams struggle when their test suite "suddenly" started taking 30+ minutes to execute and is now constantly residing in a red state. A test suite which takes longer than 30 minutes to run is a test suite no one wants to run or spend time fixing. These teams are often asking me for a single "get out of a jail card"—Page Objects or any other obscure, non-obvious testing pattern which would immediately relieve them from all their pain. The truth is much harder to swallow. In the same way it took them months to get into the situation they're in, it will take them weeks or even months to get out. And, the reason for that is the second biggest drawback of UI-obsessed Gherkin features—they are very inflexible towards their implementation.

UI Focused Scenarios are Inflexible to Implementation

The biggest benefit of using Gherkin is it allows you to separate the problem definition from the implementation, or even to a certain extent, testing. Properly written examples give you an incredible amount of flexibility around how you're going to test or implement the feature. In contrast, poorly written, obsessed with user interface examples are extremely tricky to implement or test in any other way than through the user interface.

Let's look at the first example from the beginning of this article:

```
Scenario: Product costing less than ·10 results in \
delivery cost of ·3
  Given there is a product with SKU "RS1" and a cost of \
5 in the catalog
  When I add the product with SKU "RS1" from the \
catalog to my basket
  Then the total price of my basket should be ·9
```

Do you think we can test this feature through the UI? Absolutely, we can. Do you think we are forced to? Can we test this exact scenario directly against the core domain objects, exercising them closer to the logic we care about? Yes, we can! The magic of this style of Gherkin scenarios is they allow us flexibility in their implementation. You can, for example, start by testing and implementing this particular scenario through UI with Mink. Then, when your test suite starts reaching towards a "couple of minutes per suite execution" barrier, you can go back and refactor your step definitions for this scenario to go through the business logic, separately from the infrastructure or the framework. And all without a single change to the scenario itself!

Now how about this example instead:

```
Scenario: Product costing less than ·10 results in \
delivery cost of ·3 Given there is a product with SKU \
"RS1" and a cost of 5 in the catalog
 And I am on "/catalog"
 When I press "Add RS1 to the basket"
 And I follow "My Basket"
 Then I should see "Total price: ·9"
```

Can we test this scenario against anything but UI? No, we can't. Even more worrisome is it is hard to understand what this scenario is supposed to actually test—all we see here is buttons and forms. It is famously tricky to extract the business essence from your UI scenarios. By writing your scenarios in this way you are effectively locking yourself out of other options for testing this specific feature later.

UI focused scenarios are notoriously inflexible to implementation because they are obsessed with it—pages, nodes, buttons, labels, and forms are spilled all over them. The more UI details your scenarios have, the harder it is to spot the essence of what is it you're actually trying to demonstrate here with this example—a product discount calculation.

Getting out of Jail

Imagine you're one of hundreds of teams across the world who use Behat. Let's also assume you were working on your project for quite some time and now you have hundreds of UI-obsessed scenarios and a test suite which takes 30 minutes or more to execute and is constantly broken because the Selenium is not as stable as you expect it to be. How do you get out of this?

You, and many before you (including me), skipped the most crucial part of the tool usage—the set of principles it is built upon. Remember that definition of Behavior-driven Development I gave way back? Remember the conversation in the form of examples, devoid of any particular implementation as the central piece of the BDD puzzle? Well, guess what, Behat is a BDD tool and you skipped (as did I on multiple occasions) the most important part of BDD: exploration of the problem. It is time we pay our due!

The first step in getting out of mess is, unsurprisingly, the same for any kind of software design problem—refactoring, except this time we're not talking about refactoring of our code, we're talking about refactoring of our understanding. The first thing you do is you find the most important feature file you have at this particular moment and you finally start having the conversation with your stakeholders. You come to your business experts and ask them for help; say you feel you overcomplicated some of the crucial parts of the application and it is now required for you to understand what this feature actually means:

```
Scenario: Product costing less than ·10 results in \
delivery cost of ·3
  Given there is a product with SKU "RS1" and a cost of \
·5 in the catalog
  And I am on "/catalog"
  When I press "Add RS1 to the basket"
  And I follow "My Basket"
  Then I should see "Total price: ·9"
```

Liz Keogh, a fellow BDD practitioner, uses pixies as a way to remove the implementation detail from the scenarios. What if there were no backend, frontend, PHP, JavaScript or else behind the scene? What if all your feature did was to send a written request to a bunch of pixies and they tried to fulfil it as hard as they could without any technical skills whatsoever. If you look at every single feature in your application as a black box with a bunch of magic inside (including UI), you will find very quickly it isn't hard to focus on the essence of what business wants from your application. You will quickly find that what the business you serve cares about is not this:

```
Scenario: Product costing less than ·10 results in \
delivery cost of ·3
  Given there is a product with SKU "RS1" and a cost of \
·5 in the catalog
  And I am on "/catalog"
  When I press "Add RS1 to the basket"
  And I follow "My Basket"
  Then I should see "Total price: ·9"
```

It is actually closer to this:

```
Scenario: Product costing less than ·10 results in \
delivery cost of ·3
  Given there is a product with SKU "RS1" and a cost of \
·5 in the catalog
  When I add the product with SKU "RS1" from the \
catalog to my basket
  Then the total price of my basket should be ·9
```

When going through this exercise together with your stakeholders, you quickly find the outcome is not a particularly better way to test your application or design your software. You will find the outcome is an approach which enables you to chose from more than one option. Do I hate browser automation in general or Mink in particular (even though I wrote the damn thing myself)? Of course I don't. What I do hate is the situation when the only option for your team going forward is the browser automation. Browser automation must be a deliberate and well-considered choice for every single scenario you develop. What it shouldn't be is the default for every single one of your tests.

When not blogging Konstantin Kudryashov is a prominent public speaker, organiser of BDD London meetups, the creator of Behat, Mink, co-creator of PhpSpec and leads the Behaviour-Driven Development (BDD) practice at Inviqa, a leading digital consultancy in London. As a communication coach, he has helped teams in many organisations bridge the gap between business and IT using Agile processes and development practices like Scrum, Kanban, BDD, TDD, Collaborative Product Ownership and Deliberate Discovery. @everzet

Want more articles like this one?

Keep your skills current and stay on top of the latest PHP news and best practices by reading each new issue of php[architect], jam-packed with articles.

Learn more every month about frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[architect]

magazine
books
conferences
training

www.phparch.com

Get the complete issue for only $6!

We also offer digital and print+digital subscriptions starting at $49/year.