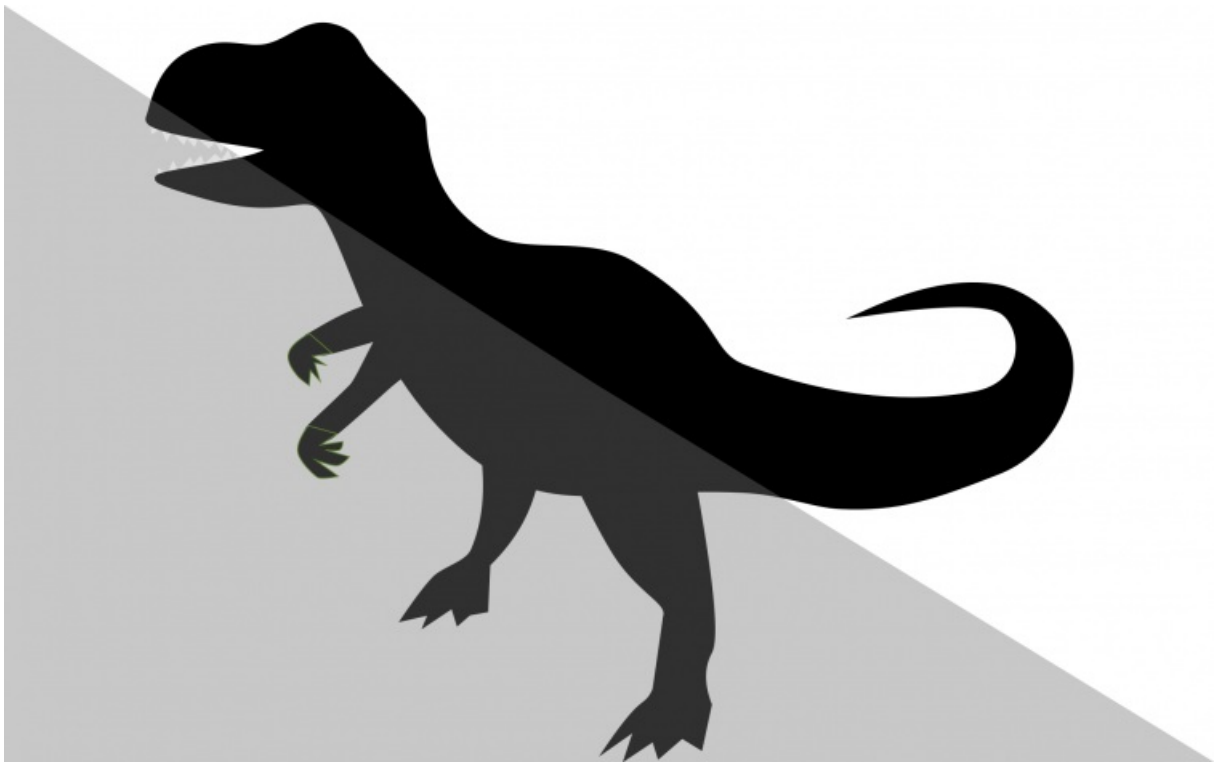
BDD, Behat, Mink and other Wonderful Things



With <3 from SymfonyCasts

Chapter 1: From Install to JS Testing

Welcome to the magical world of Behat, my favorite library. No joke this thing is the best. Behat is about two things:

BDD, Functional Testing and... Planning a Feature???

First, functionally testing your application. Which means writing code that will open a browser, fill out a form, hit submit and verify text on the other side.

Why test your application? Well, imagine you are in charge of safety at Jurassic Park, your job is to make sure guests aren't eaten by dinosaurs. You need to be certain that the new pterodactyl exhibit that's being put in won't turn off the electric fence around the velociraptor pen. No tests? Good luck, they know how to open doors.

And second, designing your application. As developers we'll often just start coding without thinking about what we're building or how the feature will behave.

Using behavior driven development, which Behat helps you do, you'll actually plan things out beforehand.

Imagine a world where communication on your team is perfect, you always deliver exactly what your client wants, electricity on the raptor fence never goes down and chocolate ice cream is always free. Yep, that's where we're going.

Behat Docs

Over in the browser, let's surf to the Behat documentation. In this tutorial we're covering version 3, and for whatever reason when I recorded this the website still defaults to version 2.5. So double check that you are actually looking at version 3's documentation.

I've got our project, the raptor store, loaded here. This is where dinosaurs go for the newest iphone and other cool electronics. There's a homepage, an admin section and that's basically it. This store is built using a very small Symfony2 application -- but hey, don't panic if you're not using Symfony: everything here will translate to whatever you're using.

Cool, we've got a search box, let's look up some sweet Samsung products. And here are the two results we have. I want to start this whole Behat thing by testing this. Hold onto your butts, let's going to get this thing running!

Install and Configuration

Over in the terminal run composer require and instead of using behat/behat we'll grab: behat/mink-extension and behat/mink-goutte-driver:

```
$ composer require behat/mink-extension behat/mink-goutte-driver
```

These are plugins for Behat and another library called Mink and they require Behat and Mink. We see the Mink library downloaded here, and the Behat library downloaded down there. So life is good!

Once you've downloaded Behat you'll have access to an executable called ./vendor/bin/behat or just bin/behat for Symfony2 users. Running it now gives us a nice strong error:

```
$ vendor/bin/behat
```

That's ok because we need to run it with --init at the end just one time in our application:

```
$ vendor/bin/behat --init
```

This did an underwhelming amount of things for us. It created two directories and one file.

In PhpStorm we see a features directory, a bootstrap directory and a little FeatureContext.php file and that's all of it:

24 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 2
use Behat\Behat\Context\Context;
use Behat\Behat\Context\SnippetAcceptingContext;
... lines 5 - 6

... lines 8 - 10
class FeatureContext implements Context, SnippetAcceptingContext
{
... lines 13 - 19
    public function __construct()
    {
    }
}
```

While we're here, I'll add a use statement for MinkContext and make it extend that. I'll explain that in a minute:

25 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 6
use Behat\MinkExtension\Context\MinkContext;

... lines 9 - 11
class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
... lines 13 - 25
```

One last bit of setup: at the root of your project create a behat.yml file. I'll paste in some content to get us started:

6 lines | behat.yml

```
default:
    extensions:
        Behat\MinkExtension:
            base_url: http://localhost:8000
            goutte: ~
```

When we run Behat it will looks for a behat.yml file and this tells it:

> Yo! Our application lives at localhost:8000, so look for it there.

Your First Feature and Scenario

Behat is installed, let's get to writing features! In the features directory create a new file called search.feature and we'll just start describing the search feature on the raptor store using a language called Gherkin which you're about to see here.

11 lines | features/search.feature

```
Feature: Search
  In order to find products dinosaurs love
  As a website user
  I need to be able to search for products
... lines 5 - 11
```

Here I'm just using human readable language to describe the search feature in general. Within each feature we'll have many different scenarios or user flows. So let's start with Scenario: Searching for a product that exists. Now using very natural language I'll describe the flow.

```
11 lines | features/search.feature
... lines 1 - 5
6   Scenario: Search for a word that exists
7     Given I am on "/"
8     When I fill in "searchTerm" with "Samsung"
9     And I press "search_submit"
10    Then I should see "Samsung Galaxy S II"
```

Don't stress out about the formatting of this, we'll cover that in detail.

The only two things that should look weird to you are searchTerm and search_submit because they are weird. searchTerm is the name attribute of this box here, and search_submit is the id of this button. We'll talk more about this later: I'm actually breaking some rules. But I want to get this working as quickly as possible.

Running Behat

Ready for me to blow your mind? Just by writing this one scenario we now have a test for our app. In the terminal run ./vendor/bin/behat and boom! It just read that scenario and actually went to our homepage, filled in the search box, pressed the button and verified that "Samsung Galaxy" was rendering on the next page. Why don't we see this happen? By default, it runs things using invisible curl request instead of opening up a real browser.

Testing JavaScript

The downside to this is that if you have JavaScript on your page that this scenario depends on, it isn't going to work since this isn't actually opening up a real browser. So, how can we run this in a real browser? There are actually a bunch of different ways. The easiest is by using Selenium.

Grab another library with composer require behat/mink-selenium2-driver. You'll also need to download the selenium server which is really easy, it's just a jar file. Click this link here under downloads to get the Selenium Standalone Server. I already have this, so I'm not actually going to download it.

To run things in Selenium, open a new tab in your terminal, and run the jar file that you just downloaded. For me that's

```
$ java -jar ~/Downloads/selenium-server-standalone-2.45.0.jar
```

Tip

Firefox 47.0.0 and lower is not supported at all since Selenium 3.0.0 - update Firefox to the latest version and install the new geckodriver for it in order to use the latest Selenium server.

This will load and run as a daemon, so it should just hang there.

Our library is done downloading and we just need to activate it in our behat.yml with the line:

```
7 lines | behat.yml
1  default:
2    extensions:
3      Behat\MinkExtension:
... lines 4 - 5
6        selenium2: ~
```

This gives me the option to use goutte to run the test using curl requests or Selenium to have things run in a browser. By default, this will just select goutte. So how do we make it use Selenium? I'm so glad you asked!

Above the scenario that you want to run in Selenium add @javascript:

12 lines | features/search.feature

```
... lines 1 - 5
6    @javascript
7    Scenario: Search for a word that exists
... lines 8 - 12
```

And that's it. Go back to the terminal and let's rerun this test. It actually opens the browser, it's quick but you can see it clicking around to complete the scenario. Cool!

Tip

FireFox is buggy with the new Selenium 3 server that's why it's preferable to use Google Chrome. You can explicitly specify the browser in the behat.yml config file:

```
# behat.yml
default:
  extensions:
    Behat\MinkExtension:
      browser_name: chrome
```

We write human readable instructions and they turn into functional tests, and this just barely scratches the surface of how this will change your development. Let's keep going and figure out what's really going on here.

Chapter 2: BDD Features

I'm sure you've heard of "Test Driven Development" where you write the tests first, and you code until those tests pass. That process is really cool! But we're talking about "Behavior Driven Development" or BDD. This is the idea that you are going to write down the behavior of your application first and then develop it.

Behavior-Driven Development

This might sound pretty obvious to you: why would I code without thinking about what I'm going to code? But that actually happens pretty often.

Behat and BDD are going to give us a very specific path to follow so that we think first, and then we code. We do this to maximize business value. There are two types of BDD, story and spec. Without getting too far into the details of each, story BDD is done with Behat and usually ends up with functional tests. Spec BDD is handled by another wonderful tool called PHPSpec and this ends up focusing on unit testing, how you actually design your classes and functions. They're both cool and in a perfect world you'll use both.

One minute of theory:

With BDD we break down our development process into four steps:

1. Define the business value of all of the big feature.
2. Prioritize those so you work on the ones with the highest business value first.
3. Take one feature and break it down into all of the different user stories or scenarios.
4. Write the code for the feature, since you now know how you want it to behave.

To do all this planning, Behat uses a language called **Gherkin**. This isn't special to Behat or PHP, it also exists in Ruby, Java and every other programming world out there which is a good thing.

Writing Features

Our application is already partially built, but let's pretend that it isn't and we're just in the planning stages. First we know we'll need an authentication feature. So let's go into the features directory and create a new authentication.feature file. Each feature will have it's own file in here:

5 lines | features/authentication.feature

```
Feature: Authentication
  In order to gain access to the site management area
  As an admin
  I need to be able to login and logout
```

We always start with the same header Feature and a short description of it. Here we'll just say "Authentication". The next three lines are very important. Our next line is always "In order to" followed by the business value. In the case of the Raptor Store, since you need to login to see the product area, I would say that the business value is "to gain access to the management area".

Next is "As a" and you say who is going to benefit from this feature, in our case it would be an admin user. And the third line is, "I need to be able to" followed by a short description of what the user would actually be able to do with this feature. In our store that is "login and logout".

The most important parts are the first two lines, the business value and the user - or role - that's going to benefit from this value. If either of these are difficult to define then maybe your feature isn't actually valuable. Or maybe none is benefiting from it. Perhaps move onto a different task.

Writing Good Business Values

Back in the Raptor store, login with admin/admin and check out the product admin area. Let's describe that! Back into the features directory and create a new product_admin.feature file. And we'll start the same as we always do:

```
5 lines | features/product_admin.feature
1  Feature: Product admin panel
   ... lines 2 - 5
```

So why do we care about having a product admin area? It's not just so we can click links and fill out boxes. Nobody cares about that. The true reason to go in there is to control the products on the frontend. So let's say just that:

```
5 lines | features/product_admin.feature
   ... line 1
2    In order to maintain the products shown on the site
3    As an admin
4    I need to be able to add/edit/delete products
```

Writing at the Tech Level of your User

That looks good. What else do we have? Check out the "Fence Security Activated" message on the site. Let's imagine we need to create an API where someone can make an API request to turn the fence security on or off from anywhere. For example, if you're running from dinosaurs somewhere, you might want to pull out your iphone and turn the fences back on.

We'll need another feature file called fence_api.feature. Start with:

```
5 lines | features/fence_api.feature
1  Feature: Remote control fence API
   ... lines 2 - 5
```

The business value is pretty clear:

```
5 lines | features/fence_api.feature
   ... line 1
2    In order to control fence security from anywhere
   ... lines 3 - 5
```

The user that benefits from this isn't some browser-using admin user: it's a more-advanced API user:

```
5 lines | features/fence_api.feature
   ... lines 1 - 2
3    As an API user
4    I need to be able to POST JSON instructions that turn fences on/off
```

I feel safer already.

Bad Business Value

There's a common pitfall, and it looks like this:

```
5 lines | features/product_admin.feature
   ... line 1
2    In order to add/edit/delete products
3    As an admin
4    I need to be able to add/edit/delete products
```

Notice the In order to and the I need to be able to lines are basically the same. This is a sign of a bad business value. Being able to add/edit/delete products is not a business value. People don't go into the product admin area just for the delight of adding, editing and deleting products. They go into the admin area because that allows them to control the products that are rendered on the front end.

This is subtle, but really important because when we build the admin area, it will focus our efforts: we know that we're building this just as a tool so that you can control things on the frontend. Which sadly for the developer means we maybe don't need that crazy drag-and-drop interface for adding videos from YouTube. Our admin user - who we're building this

feature for - doesn't care about using it, and it might be too technical for them anyways.

Focus on your business value, and keep the feature at the technical level of who you're building it for.

Chapter 3: Scenarios

We've created our features, which is really nice because we can see the business value of each and decide which one to start on first. I want to start with our product admin feature, so our admin users can start loading in data.

Now that we've selected that, we'll start adding scenarios to the feature which are essentially user stories. My first scenario is going to say, "when I go to the admin area and click on products, I see the products listed."

So let's start with:

20 lines | features/product_admin.feature

```
... lines 1 - 5
6     Scenario: List available products
... lines 7 - 20
```

Do keep in mind that this line doesn't matter too much, no need to be Shakespeare.

Given, When and Then

For these scenarios I need you to think as if you *are* the admin user. We'll talk in the first person point of view at the intended user's technical level. Meaning, clicking on buttons an admin user can actually recognize and viewing things that they will see. Put yourself in their shoes.

Given: Setup

Scenarios always have three parts, the first is Given where you can create any type of setup before the user story starts. So in our case, if we want to list products, we need to make there are some in the database to see:

20 lines | features/product_admin.feature

```
... lines 1 - 6
7     Given there are 5 products
... lines 8 - 20
```

Why am I saying exactly "there are 5 products"? Great question, the exact phrasing here doesn't matter at all. I'm just using whatever language sounds most clear to me. Move on down to the next line:

20 lines | features/product_admin.feature

```
... lines 1 - 7
8     And I am on "/admin"
... lines 9 - 20
```

Using the word And here extends the Given line to include more setup details. With these two lines, we'll have 5 products in the database and start the test on the /admin page.

When: User Action

The second part of every scenario is When which is the user action. Here, "I" - the admin user - actually take action. In our case, the only action is clicking products:

20 lines | features/product_admin.feature

```
... lines 1 - 8
9     When I click "Products"
... lines 10 - 20
```

This is good because that's the name of the link and it reads very clearly.

User Expectations

Finally, the last part of every scenario is Then, this is where we witness things as the user. In our scenario, we should see the 5 products that we set up in our Given:

```
20 lines | features/product_admin.feature
... lines 1 - 9
10      Then I should see 5 products
... lines 11 - 20
```

Always use language that is clear to you after your Given, When, Then or And in your scenarios.

Only play God in Given

In Given you can add things to the database before hand, but once you are in When and Then you should only be taking actions that the admin user could take and viewing things they can view. This means we won't be using CSS selectors in our scenarios or phrasing such as "Then a product should be entered into the database" because a user can't see that happen. However, the user could see a helpful message, like "Celebrate, your product was saved!"

BDD with a New Scenario

Let's create a new scenario for functionality that doesn't actually exist yet: adding a new product. Time to plan out this scenario!

```
20 lines | features/product_admin.feature
... lines 1 - 11
12    Scenario: Add a new product
... lines 13 - 20
```

We don't need to add any products in the database before hand so we just start with:

```
20 lines | features/product_admin.feature
... lines 1 - 12
13      Given I am on "/admin/products"
... lines 14 - 20
```

We're going to want a button on this page that says "New Product". So let's add clicking that to the process of creating a new product:

```
20 lines | features/product_admin.feature
... lines 1 - 13
14      When I click "New Product"
... lines 15 - 20
```

Clicking this will take us to another page with a form containing fields for name, price and description. To keep our When going to include the action of filling out this form we'll use And:

```
20 lines | features/product_admin.feature
... lines 1 - 14
15      And I fill in "Name" with "Veloci-chew toy"
16      And I fill in "Price" with "20"
17      And I fill in "Description" with "Have your velociraptor chew on this instead!"
... lines 18 - 20
```

There *is* a secret reason of why I'm using language like "I fill in" after the And, which we'll cover soon. Lastly, in our process I'll expect to hit a save button to finish creating my new product. So let's add another line to our scenario:

20 lines | features/product_admin.feature

```
... lines 1 - 17
18      And I press "Save"
... lines 19 - 20
```

At this point my product should be saved and I will be redirected to another page with a success message to prove that this worked. This is where we'll say:

20 lines | features/product_admin.feature

```
... lines 1 - 18
19      Then I should see "Product created FTW!"
```

Isn't that a nice way to plan out your user flow? When we *do* finally code this, we'll know exactly how everything should work.

Chapter 4: Behat

Ok, the basics of Gherkin, writing features and scenarios, are now behind us. Now, how does this "Behat" thing fit in?

Imagine we've gone back in time 25 years and Linus Torvalds, the Yoda of Linux, comes to us and says:

> I would love your help in building the ls command.

Yep, the ls command right here in the terminal. Since you're an awesome dev, you reply:

> I would be happy to help you Linus, and I'll use Gherkin to describe the feature and the scenarios for the ls command. I'm really into Behavior Driven Development.

Writing Scenario #1

Create a new ls.feature file:

12 lines | features/ls.feature

```
Feature: ls
  In order to see the directory structure
  As a UNIX user
  I need to be able to list the current directory's contents
... lines 5 - 12
```

I'll save some time by copying the feature description in since we've already covered this pretty well. On to the scenario! The first scenario might be:

12 lines | features/ls.feature

```
... lines 1 - 5
  Scenario: List 2 files in a directory
... lines 7 - 12
```

And now we'll go through our Given, When, and Then lines. Since we need to list two files we'll need to create those first:

12 lines | features/ls.feature

```
... lines 1 - 6
    Given I have a file named "john"
    And I have a file named "hammond"
... lines 9 - 12
```

The user action would be actually running the ls command:

12 lines | features/ls.feature

```
... lines 1 - 8
    When I run "ls"
... lines 10 - 12
```

Finally, we'll actually test that the "john" and "hammond" files both appear in the output:

12 lines | features/ls.feature

```
... lines 1 - 9
    Then I should see "john" in the output
    And I should see "hammond" in the output
```

Writing this scenario has two purposes. First, it lets us plan how our feature should behave. And that's what we've been

talking about. Second, we want the scenario to be executed as a test to prove whether or not we have successfully created this behavior. To do that, we'll run ./vendor/bin/behat in our terminal and point it at features/ls.feature and let's see what happens!

The Essence of Behat: Matching Scenario lines with Functions

Ahh, so it says our scenario was undefined and something about our FeatureContext having missing steps. And it even gives us some PHP code. Copy these three functions, open up the FeatureContext class that we generated and paste them there:

49 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 11
class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
{
... lines 14 - 24
    /**
     * @Given I have a file named :arg1
     */
    public function iHaveAFileNamed($arg1)
    {
        throw new PendingException();
    }

    /**
     * @When I run :arg1
     */
    public function iRun($arg1)
    {
        throw new PendingException();
    }

    /**
     * @Then I should see :arg1 in the output
     */
    public function iShouldSeeInTheOutput($arg1)
    {
        throw new PendingException();
    }
}
```

So what does Behat really do? Simply, it reads each line of our scenario, looks for a matching function inside of FeatureContext and calls it. In this case, it'll read "There is a file named "john", find that it matches this annotation here and then execute its function.

What's really cool is that because we surrounded john with quotes, it recognized that as a wildcard. In the annotation, we have :arg1 which means it matches anything surrounded in quotes or a number.

Filling in the Definitions/Functions

Our job is just to make these functions do what they say they will do. I'll change this to :filename and update the argument to $filename, just because that's more descriptive:

49 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 11
class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
{
... lines 14 - 24
    /**
     * @Given I have a file named :filename
     */
    public function iHaveAFileNamed($filename)
... lines 29 - 47
}
```

In this function, how do we create a file? How about we use touch($filename);:

49 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 29
        touch($filename);
... lines 31 - 49
```

For iRun() update the arg1's to command. There are lots of ways to run commands, but we'll use shell_exec($command);:

49 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 32
    /**
     * @When I run :command
     */
    public function iRun($command)
    {
        shell_exec($command);
    }
... lines 40 - 49
```

Sharing Data inside your Scenario (between "Steps")

Lastly, in iShouldSeeInTheOutput(), update the arg1 to string:

53 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 42
    /**
     * @Then I should see :string in the output
     */
    public function iShouldSeeInTheOutput($string)
... lines 47 - 53
```

And now we're stuck... we don't have the return value from shell_exec() above. Good news, there is a really nice trick for this. Whenever you need to share data between functions in FeatureContext, you'll just create a new private property. At the top of the class let's create a private $output property and update the iRun function to $this->output = shell_exec($command);:

53 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 11
class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
{
    private $output;
... lines 15 - 37
    public function iRun($command)
    {
        $this->output = shell_exec($command);
    }
... lines 42 - 51
}
```

Behat doesn't care about the new property: this is just us being good object oriented programmers and sharing things inside a class.

Lifetime of a Scenario

This works because every scenario gets its own FeatureContext object. When we have more scenarios later, Behat will instantiate a fresh FeatureContext object before calling each one. So we can set any private properties that we want on top, and only just this scenario will have access to it.

Failing!

Now in iShouldSeeInTheOutput() method if (strpos($this->output, $string) === false)) then we have a problem and we want this step to fail:

53 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 45
    public function iShouldSeeInTheOutput($string)
    {
        if (strpos($this->output, $string) === false) {
... line 49
        }
    }
... lines 52 - 53
```

How do you fail in Behat? By throwing an exception: throw new \Exception() and print a really nice message here of "Did not see '%s' in the output '%s'. Finish that line up with $string, $this->output:

53 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 48
            throw new \Exception(sprintf('Did not see "%s" in output "%s"', $string, $this->output));
... lines 50 - 53
```

Ok let's give this a try!

Re-run our last Behat command in the terminal, and this time it's green! And if you run the ls command here you can see the "john" and "hammond" files listed.

What are Definitions and Steps?

Back to our scenario. Get out some pen and paper: we need to review some terminology! Every line in here is called a "step":

12 lines | features/ls.feature

```
... lines 1 - 6
7       Given I have a file named "john"
8       And I have a file named "hammond"
9       When I run "ls"
10      Then I should see "john" in the output
11      And I should see "hammond" in the output
```

And the function that a step connects to is called a "step definition":

53 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 11
12  class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
13  {
... lines 14 - 26
27      /**
28       * @Given I have a file named :filename
29       */
30      public function iHaveAFileNamed($filename)
... lines 31 - 34
35      /**
36       * @When I run :command
37       */
38      public function iRun($command)
... lines 39 - 42
43      /**
44       * @Then I should see :string in the output
45       */
46      public function iShouldSeeInTheOutput($string)
... lines 47 - 51
52  }
```

This is important in helping you understand Behat's documentation.

Oh, and the 4 feature lines above: those aren't parsed by Behat:

12 lines | features/ls.feature

```
1   Feature: ls
2     In order to see the directory structure
3     As a UNIX user
4     I need to be able to list the current directory's contents
... lines 5 - 12
```

We only write those to go through the exercise of thinking about our business value.

Now, you see in our test it says "5 steps (5 passed)", which means that each step could fail. The rule is really simple, if the definition function for a step throws an exception, it's a failure. If there's no exception it passes.

Head back to the step that looks for the "hammond" file and add the number 2 to the end of the file name so it fails. Running the scenario in our terminal now shows us 4 steps passed and 1 failed.

Chapter 5: Behat Hooks Background

Behat experts coming through! Seriously, we've covered it: Behat reads the steps, finds a function using this nice little pattern, calls that function, and then goes out to lunch. That's all that Behat does, plus a few nice extras.

Let's dive into some of those extras! This scenario creates the "john" and "hammond" files inside this directory but doesn't even clean up afterwards. What a terrible roommate.

Let's first put these into a temporary directory. We'll use the __construct function because that's called before each scenario. Type mkdir('test'); and chdir('test');:

55 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 11
12  class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
13  {
... lines 14 - 22
23      public function __construct()
24      {
25          mkdir('test');
26          chdir('test');
27      }
... lines 28 - 53
54  }
```

Over in the terminal, delete the "john" and "hammond" files so we can have a fresh start at this. Rerun Behat for our ls scenario:

```
$ vendor/bin/behat features/ls.feature
```

Everything still passes and hey look there's a little test/ directory and the "john" and "hammond" are inside of that. Cool.

Ready for the problem? Rerun that test one more time. Now, errors show up that say:

> mkdir(): file exists.

This error didn't break our test but it does highlight the problem that we don't have any cleanup. After our tests run these files stick around.

We need to run some code after every scenario. Behat has a system called "hooks" where you can make a function inside of your context and tell Behat to call it before or after your scenario, entire test suite or individual steps.

Create a new public function inside called public function moveOutOfTestDir():

75 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 11
12  class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
13  {
... lines 14 - 66
67      public function moveOutOfTestDir()
68      {
... lines 69 - 72
73      }
74  }
```

This will be our cleanup function. Use chdir('..'); to go up one directory. Then, if the test/ directory exists - which it should - then we'll run a command to remove that:

75 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 68
        chdir('..');
        if (is_dir('test')) {
            system('rm -r '.realpath('test'));
        }
... lines 73 - 75
```

To get Behat to actually call this after every scenario, add an @AfterScenario annotation above the method:

75 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 63
    /**
     * @AfterScenario
     */
    public function moveOutOfTestDir()
... lines 68 - 75
```

That's it!

Let's give this a try:

```
$ vendor/bin/behat features/ls.feature
```

The first time we run this we still get the warning since our clean up function hasn't been called yet. But when we run it again, the warnings are gone! And if we run ls, we see that there is no test directory.

We can do this same thing with the mkdir(); and chdir(); stuff. Create a new public function moveIntoTestDir():

75 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 11
class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
{
... lines 14 - 55
    public function moveIntoTestDir()
    {
... lines 58 - 61
    }
... lines 63 - 73
}
```

And we can make it even a bit more resistant by checking to see if the test directory is already there and only create it if we need to. Above this, add @BeforeScenario:

75 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 52
    /**
     * @BeforeScenario
     */
    public function moveIntoTestDir()
    {
        if (!is_dir('test')) {
            mkdir('test');
        }
        chdir('test');
    }
... lines 63 - 75
```

This is basically the same as putting the code in __construct(), but with some subtle differences. @BeforeScenario is the proper way to do this.

When we run things now, everything looks really nice. I think this ls command is going to be a success!

PHPUnit Assert Functions

So bonus feature #1 is the hook system. And bonus feature #2, has nothing to do with Behat at all. It actually comes from PHPUnit. Our first step will be to install PHPUnit with composer require phpunit/phpunit --dev. That will add it under a new require-dev section in composer.json:

29 lines | composer.json

```
{
... lines 2 - 21
    "require-dev": {
... lines 23 - 25
        "phpunit/phpunit": "^4.8"
    }
}
```

Full disclosure, I should have put all the Behat and Mink stuff inside of the require-dev too: it is a better place for it since we only need them while we're developing.

I installed PHPUnit because it has really nice assert functions that we can get a hold of. To get access to them we just need to add a require statement in our FeatureContext.php file, require_once then count up a couple of directories and find vendor/phpunit/phpunit/src/Framework/Assert/Functions.php:

79 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 8
require_once __DIR__.'/../../vendor/phpunit/phpunit/src/Framework/Assert/Functions.php';
... lines 10 - 79
```

Requiring this file gives you access to all of PHPUnit's assert functions as flat functions. Down in the iShouldSeeInTheOutput() method, use assertContains(), give it the needle which is $string and the haystack which is $this->output. Finally, add our helpful message which I'll just cut and paste. Remove the rest of the original if statement:

79 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 13
class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
{
... lines 16 - 44
    /**
     * @Then I should see :string in the output
     */
    public function iShouldSeeInTheOutput($string)
    {
        assertContains(
            $string,
            $this->output,
            sprintf('Did not see "%s" in output "%s"', $string, $this->output)
        );
    }
... lines 56 - 77
}
```

Run the test again!

```
$ vendor/bin/behat features/ls.feature
```

Beautiful, it looks just like it did before.

Using Background

To show you the final important extra for Behat, create another scenario for Linus' ls feature. This time we'll say:

19 lines | features/ls.feature

```
... lines 1 - 12
13    Scenario: List 1 file and 1 directory
... lines 14 - 19
```

I'll copy all the steps from our first scenario and just edit the second line to:

19 lines | features/ls.feature

```
... lines 1 - 13
14    Given I have a file named "john"
15    And I have a dir named "ingen"
16    When I run "ls"
17    Then I should see "john" in the output
... lines 18 - 19
```

And update the final line to:

19 lines | features/ls.feature

```
... lines 1 - 17
18    And I should see "ingen" in the output
```

Man what a great looking scenario, let's run it!

```
$ vendor/bin/behat features/ls.feature
```

As expected it now says there's one missing step definition. Copy the PHP code that prints out into FeatureContext. Remove the throw exception line, and update the arg1's to dir:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 13
14  class FeatureContext extends MinkContext implements Context, SnippetAcceptingContext
15  {
... lines 16 - 78
79      /**
80       * @Given I have a dir named :dir
81       */
82      public function iHaveADirNamed($dir)
83      {
... line 84
85      }
86  }
```

Inside the function use mkdir($dir) to actually make that directory:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 83
84          mkdir($dir);
... lines 85 - 87
```

Simple!

Back to the terminal to rerun the tests. It works! And that was easy. Once you're done celebrating you may start to notice the

duplication we have in the scenarios. There are two ways to clean this up. The most important way is with Background::

```
20 lines | features/ls.feature
... lines 1 - 5
6   Background:
7     Given I have a file named "john"
... lines 8 - 20
```

If every single scenario in your feature starts with the same lines then you should move that up into a new Background section.

Now, I'll change the first line of And in each of these scenarios to Given:

```
20 lines | features/ls.feature
... lines 1 - 8
9   Scenario: List 2 files in a directory
10    Given I have a file named "hammond"
... lines 11 - 13
14
15  Scenario: List 1 file and 1 directory
16    Given I have a dir named "ingen"
... lines 17 - 20
```

I don't have to do this, but it reads better to me. Now Behat will run that Background line before each individual scenario and you'll even see that:

```
$ vendor/bin/behat features/ls.feature
```

The Background is read up here, but it actually is running before the top scenario and the bottom one. We know this because if it didn't, these tests wouldn't be passing.

Second, when you have duplication that's not on the first line of all of your scenarios like the "Then I should see...." you may want to use scenario outlines. It's a little less commonly used but we'll dive into that a bit later.

Ok, not only do you know how Behat works but you even know all of its top extra features -- check you out!

Chapter 6: Mink

Forget about Behat for a few minutes - I want to talk about a totally different library. Imagine if there were a simple PHP library that let you take command of a browser: surf to pages, find links with CSS, click those links, fill out forms and *anything* else you can dream up. It exists! And it's called Mink.

This is such a cool library that it deserves some direct attention. Start by creating a mink.php file right at the root of your project. And require composer's autoload file:

```
5 lines | mink.php
... lines 1 - 2
require __DIR__.'/vendor/autoload.php';
... lines 4 - 5
```

We'll use Mink all by itself, outside of Symfony, Behat and everything else so we can focus on just how *it* works.

Important Object 1: The Driver

As awesome as Mink is, it only has four important objects. The first is the driver. Create a new $driver variable and set it to new GoutteDriver():

```
9 lines | mink.php
... lines 1 - 4
use Behat\Mink\Driver\GoutteDriver;

... line 7
$driver = new GoutteDriver();
```

Now, ignore it: I'll come back and explain important object number 1 in a little while.

Important Object 2: The Session

Let's move on to important object #2, and the first that we really care about: the session. Add $session = new Session() and pass it the $driver as an argument:

```
19 lines | mink.php
... lines 1 - 5
use Behat\Mink\Session;

... lines 8 - 11
$session = new Session($driver);
... lines 13 - 19
```

Think of the session like a browser tab: anything you can do in a tab, you can do in a session. And actually, that isn't very much. You can visit URLs, refresh, go backwards, go forwards and that's about it. Let's use it to visit a very awesome and absurdly-designed site "jurassicpark.wikia.com".

Tip

After vocal protests from the dinosaurs, jurassicpark.wikia.com was redesigned! That means its HTML/CSS was updated after we recorded this screencast. Don't worry: we've updated the code in the code blocks on this page to work for the new design. Use them instead of the CSS selectors used in the video.

After that we'll just print out a few things about the page like the status code, and the current URL:

19 lines | mink.php

```
... lines 1 - 14
$session->visit('http://jurassicpark.wikia.com');

echo "Status code: ". $session->getStatusCode() . "\n";
echo "Current URL: ". $session->getCurrentUrl() . "\n";
```

To execute this, head over to the terminal and run:

```
$ php mink.php
```

Look at that: it's printing out 200 and a slightly different URL than we put in our code. That makes sense: when you go to the site in a browser, it redirects to the URL we see in our terminal. Mink is emulating a real browser by following redirects. But in reality, so far, it's making invisible cURL requests: it's not using a real browser.

Important Object 3: The Page (DocumentElement)

The third important object is called the page. Grab it by saying $page = $session->getPage():

24 lines | mink.php

```
... lines 1 - 20
$page = $session->getPage();
... lines 22 - 24
```

I want you to think of this as the JQuery object or the DOM. Anything you can do with JQuery - like select elements, click links and fill out fields - you can do with the page. Less impressively, it also knows the HTML of whatever page we're currently on.

Tip

If you like to dig into the source code, the page is an instance of DocumentElement.

Let's use it to print out this first bit of text on the page with var_dump(substr($page->getText()), 0, 75);:

24 lines | mink.php

```
... lines 1 - 22
echo "First 75 chars: ".substr($page->getText() , 0, 75) . "\n";
```

Run that again in the terminal.

```
$ php mink.php
```

Now we see the thrilling text of: "Park Pedia - Jurassic Park, Dinosaurs, Stephen Spielberg...". There's some weird a:lang code stuff on the end. Open up the source on the page.

The getText() method returns anything *other* than the HTML tags themselves. The first part comes from the title tag and then it grabs some other stuff from the style tag, which is technically text.

But what we *really* want to do is find individual elements so we can click links and fill out fields. Let's talk about that next.

Chapter 7: Finding Elements by CSS and Name

Tip

This element *was* an h1 tag (when we recorded the video) - now the site name is in a tag which is inside .wds-community-header__sitename block. The code below was updated.

Back on the weird Wiki page, run an inspect element on the navigation. There's a hidden h2 tag inside of the WikiHeader and WikiNav elements. Let's try to find this and print out its text.

Finding Elements by CSS

To do that use the find() function: pass it css as the first argument and then use your css selector: .WikiHeader .WikiNav h2:

33 lines | mink.php

```
... lines 1 - 25
$header = $page->find('css', '.wds-community-header__sitename a');
... lines 27 - 33
```

Important Object 4: An Element (NodeElement)

Surprise! This is actually the fourth - and final - important object in Mink. You start with the page, but as soon as you find a single element, you now have a NodeElement. What's cool is that this new object has *all* the same methods as the page, plus a bunch of extras that apply to individual elements.

Let's dump the $header->getText();:

33 lines | mink.php

```
... lines 1 - 27
echo "The wiki site name is: ".$header->getText()."\n";
... lines 29 - 33
```

And re-run the mink file. Now it prints "Jurassic Park Wiki Navigation" - so finding by CSS is working.

Finding an Element, then Finding Deeper

Let's do something harder and see if we can find the "wiki activity" link in the header by drilling down into the DOM twice. First, find the parent element by using its subnav-2 class. So I'll say $page->find('css', '.subnav-2');. Oh and don't forget your dot!

33 lines | mink.php

```
... lines 1 - 29
$subNav = $page->find('css', '.wds-tabs');
var_dump($subNav->getHtml());
... lines 32 - 33
```

Now, var_dump() this element's HTML to make sure we've got the right one. Run mink.php:

```
$ php mink.php
```

Great - it prints out all the stuff inside of that element, including the WikiActivity link that we're after.

To find *that*, we need to find the li and a tags that are inside of the .subnav-2. We could do that by just modifying the original selector. But instead, once you have an individual element you can use find() again to look inside of *it*. So we can say $nav->find() and use css to go further inside of it with li a:

35 lines | mink.php

```
... lines 1 - 29
$subNav = $page->find('css', '.wds-tabs');
$linkEl = $subNav->find('css', 'li a');
... lines 32 - 35
```

The find() method returns the *first* matching element.

Dump this element's text and check things:

35 lines | mink.php

```
... lines 1 - 32
echo "The link text is: ". $linkEl->getText() . "\n";
... lines 34 - 35
```

Yes! It returns Wiki Activity!

Find via the Amazing Named Selector

In addition to CSS, there's one more important way to find things using Mink: it's called the named selector. I'm going to paste in some code here: please do not write this -- it's ugly code -- I'll show you a better way.

Instead of passing css to find(), this passes named along with an array that says we're looking for a "link" whose text is "Wiki Activity". The named selector is all about finding an element by its visible text. To see if this is working let's var_dump($linkEl->getAttribute('href'));:

43 lines | mink.php

```
... lines 1 - 32
$selectorsHandler = $session->getSelectorsHandler();
$linkEl = $page->find(
    'named',
    array(
        'link',
        $selectorsHandler->xpathLiteral('Books')
    )
);

echo "The link href is: ". $linkEl->getAttribute('href') . "\n";
```

That should come back as the URL to the activity section. Try it out.

```
$ php mink.php
```

It works! The named selector is *hugely* important because it lets us find elements by their natural text, instead of technical CSS classes. In this case, we're using the text of the anchor tag. But the named selector also looks for matches on the title attribute, on the alt attribute of an image inside of a link and several other things. It finds elements by using anything that a user or a screen reader thinks of as the "text" of an element.

And instead of using this big ugly block of code, you'll use the named selector via $page->findLink(). Pass it "Wiki Activity":

36 lines | mink.php

```
... lines 1 - 32
$linkEl = $page->findLink('Books');

echo "The link href is: ". $linkEl->getAttribute('href') . "\n";
```

This should work just like before.

Named Selector: Links, Fields and Buttons

The named selector can find 3 different types of elements: links, fields and buttons. To find a field, use $page->findField(). This works by finding a label that matches the word "Description" and *then* finds the field associated to that label. To find a button, use $page->findButton(). Oh, and the named selector is "fuzzy" - so it'll match just *part* of the text on a button, field or link.

Click that Link Already!

Ok! Let's finally click this link! Once you have a NodeElement, just use the click() method:

```
39 lines | mink.php
... lines 1 - 36
37 $linkEl->click();
38 echo "Page URL after click: ". $session->getCurrentUrl() . "\n";
```

Run the script:

```
$ php mink.php
```

You can see it pause as it clicks the link and waits for the next page to load. And then it lands on the Special:WikiActivity URL.

dragTo, blur, check, setValue and Other Things you can do with a Field

When you have a single element, there are *a lot* of things you can do with it, and each is a simple method call. We've got focus, blur, dragTo, mouseOver, check, unCheck, doubleClick and pretty much everything you can imagine doing to an element.

GoutteDriver = cURL, Selenium2Driver = Real Browser

Head back up to the GoutteDriver part - that was important object number 1. The driver is used to figure out *how* a request is made. The Goutte driver uses cURL. If we wanted to use Selenium instead, we only need to change the driver to $driver = new Selenium2Driver();:

```
42 lines | mink.php
... lines 1 - 9
10 //$driver = new GoutteDriver();
11 $driver = new Selenium2Driver();
... lines 12 - 42
```

Tip

By default, the Selenium2 driver uses Firefox. But recent versions may not work correctly with Selenium server. If you have any issues, try using Google Chrome instead:

```
// ...
$driver = new Selenium2Driver('chrome');
```

That's it! Oh and make sure you have $session->start() at the top:

```
42 lines | mink.php
... lines 1 - 15
16 $session->start();
... lines 17 - 42
```

I should have had this before, but Goutte doesn't require it. Similarly, at the bottom, add $session->stop();:

```
42 lines | mink.php
... lines 1 - 41
42 $session->stop();
```

That closes the browser.

In our terminal, I still have the Selenium JAR file running in the background. Run php mink.php.

The browser opens... but just hangs. Check out the terminal. It died!

> Status code is not available from Behat\Mink\Driver\Selenium2Driver.

The cause is the $session->getStatusCode() line:

39 lines | mink.php

```
... lines 1 - 16
17 echo "Status code: ". $session->getStatusCode() . "\n";
18 echo "Current URL: ". $session->getCurrentUrl() . "\n";
... lines 19 - 39
```

Different drivers have different super powers. The Selenium2 driver can process JavaScript: a pretty sweet super power. But it also has its own weakness, like kryptonite and the inability to get the current status code.

The driver you'll use depends on what functionality you need, which is why Mink made it so easy to switch from one driver to another. Remove the getStatusCode() line and re-run the script:

42 lines | mink.php

```
... lines 1 - 18
19 //echo "Status code: ". $session->getStatusCode() . "\n";
20 echo "Current URL: ". $session->getCurrentUrl() . "\n";
... lines 21 - 42
```

Other than this annoying FireFox error I started getting today, it works fine. The browser closes, and we're now dangerous with Mink.

Let's put this all together!

Chapter 8: Behat Loves Mink (Free Definitions from MinkExtension)

Behat parses scenarios and Mink is really good at browsing the web. If we combine their powers, we could start having steps that look a lot like what we have in search.feature.

11 lines | features/search.feature

```
... lines 1 - 5
6     Scenario: Search for a word that exists
7       Given I am on "/"
8       When I fill in "searchTerm" with "Samsung"
9       And I press "search_submit"
10      Then I should see "Samsung Galaxy S II"
```

For the Background step, we *now* know we could create a matching definition in FeatureContext and easily use Mink's session object to actually go to that URL. But earlier when we ran this scenario, it worked... so there must already be something tie'ing Behat and Mink together.

Let's see what's happening.

Free Behat Steps

First, in FeatureContext I had you extend MinkContext. Remove that now:

86 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 12
13  class FeatureContext implements Context, SnippetAcceptingContext
... lines 14 - 86
```

When we run Behat, it needs to know *all* of the step definition language that's available. You can see that list by passing a -dl to the Behat command:

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

This shows the four ls definitions we built. So, Behat opens the FeatureContext class, parses out all of the @Given, @When and @Then annotations, and prints a final list here for our enjoyment.

When we add more step definitions, this list grows. And if we use something that *isn't* here yet, Behat very politely prints out the function for us in the terminal.

In behat.yml we added this MinkExtension configuration:

12 lines | behat.yml

```
1   default:
... lines 2 - 6
7       extensions:
8           Behat\MinkExtension:
... lines 9 - 12
```

This library ties Behat and Mink together and gives us two cool things. First, it lets us access the Mink Session object inside of FeatureContext. We'll see that soon.

For the second thing, add a new config called suites: and a key under that called default: with a contexts: key. We'll talk about suites later. Under contexts, pass FeatureContext *and* Behat\MinkExtension\Context\MinkContext:

12 lines | behat.yml

```
default:
    suites:
        default:
            contexts:
                - FeatureContext
                - Behat\MinkExtension\Context\MinkContext
... lines 7 - 12
```

Now, Behat will look inside FeatureContext *and* MinkContext for those definition annotations.

Let's see what that gives us: run behat with the -dl option again:

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

Boom! Now we see a *huge* list! These include definitions for all common web actions, like When I go to or When I fill in "field" with "value". This includes the stuff we're using inside of search.feature:

11 lines | features/search.feature

```
... lines 1 - 5
    Scenario: Search for a word that exists
      Given I am on "/"
      When I fill in "searchTerm" with "Samsung"
      And I press "search_submit"
      Then I should see "Samsung Galaxy S II"
```

So *that's* why that scenario already worked.

Let's take a look at where these come from. I'll use shift+shift and search for MinkContext:

487 lines | vendor/behat/mink-extension/src/Behat/MinkExtension/Context/MinkContext.php

```
... lines 1 - 21
class MinkContext extends RawMinkContext implements TranslatableContext
{
    /**
... lines 25 - 26
     * @Given /^(?:|I )am on (?:|the )homepage$/
     * @When /^(?:|I )go to (?:|the )homepage$/
     */
    public function iAmOnHomepage()
    {
        $this->visitPath('/');
    }
... lines 34 - 75
    /**
... lines 77 - 78
     * @When /^(?:|I )press "(?P<button>(?:[^"]|\\")*)"$/
     */
    public function pressButton($button)
    {
        $button = $this->fixStepArgument($button);
        $this->getSession()->getPage()->pressButton($button);
    }
... lines 86 - 485
}
```

This looks just like our FeatureContext, but has a bunch of goodies already filled in.

So, why did I use this exact language inside of my scenario originally? Because, I'm lazy, and I knew if I followed the language here, I'd get all this functionality for free. And I'm from the midwest in the US: we love free things.

I'll take off the @javascript line:

11 lines | features/search.feature

```
... lines 1 - 4
5
6     Scenario: Search for a word that exists
... lines 7 - 11
```

Since we don't need JavaScript, and now we should be able to run our search feature. Perfect!

Chapter 9: Scenario Outline

Add a second scenario to search: searching for a product that is *not* found. I'm lazy, so I'll copy the first scenario.

18 lines | features/search.feature

```
... lines 1 - 13
14    Scenario: Search for a word that does not exist
... lines 15 - 18
```

The Given is the same in both, so we can move that to a Background:

18 lines | features/search.feature

```
... lines 1 - 5
6     Background:
7       Given I am on "/"
... lines 8 - 18
```

It *is* ok not to have any Given's in the scenario and start directly with When. I'll change the search term to "XBox" - none of those in the dino store. This is really going to bum out any dino's that are looking to play Ghost Recon.

When there are no matches - the site says "No Products Found". Cool, let's plug that into our Then line:

18 lines | features/search.feature

```
... lines 1 - 13
14    Scenario: Search for a word that does not exist
15      When I fill in "searchTerm" with "XBox"
16      And I press "search_submit"
17      Then I should see "No products found"
```

Run this:

```
$ ./vendor/bin/behat
```

We didn't use any new language, so everything runs *and* passes!

Scenario Outlines

To remove duplication, we can take things a step further with a Scenario Outline. It looks like this. Copy one of the scenarios and change it to start with Scenario Outline:.

The search term is different in the scenarios, so change it to <term>. For the "should see" part, replace that with <result>:

18 lines | features/search.feature

```
... lines 1 - 8
9     Scenario Outline: Search for a product
10      When I fill in "searchTerm" with "<term>"
11      And I press "search_submit"
12      Then I should see "<result>"
... lines 13 - 18
```

Now, we have two variable parts. To fill those in, add an Examples section at the end with a little table that has term and result columns. For the first row, use Samsung for the term and Samsung Galaxy for the result. The second row should be XBox and the No products found message:

18 lines | features/search.feature

```
... lines 1 - 13
14      Examples:
15        | term    | result              |
16        | Samsung | Samsung Galaxy S II |
17        | XBox    | No products found   |
```

Now we remove both of the scenarios -- crazy right? You'll see this table format again later. The table doesn't need to be pretty, but I like to reformat it with cmd + option + L.

Time to try this out.

Perfect! It only prints out the Scenario Outline once. But below that, Behat prints both examples in green as it executes each of them. Senario Outlines are the best way to reduce duplication that goes past just the first Given line.

Chapter 10: Mink Session inside FeatureContext

In search.feature:

18 lines | features/search.feature

```
... lines 1 - 9
10      When I fill in "searchTerm" with "<term>"
11      And I press "search_submit"
... lines 12 - 18
```

This searchTerm is the name attribute of the search box. And search_submit is the id of its submit button. Well, listen up y'all, I'm about to tell you one of the most important things about working with Behat: Almost every built-in definitions finds elements using the "named" selector, *not* CSS.

Cardinal Rule: Avoid CSS in your Scenarios

For example, look at the definition for

> I fill in "field" with "value"

To use this, you should pass the *label* text to "field", *not* the id, name attribute or CSS selector. Clicking a link is the same. That's done with the

> I follow "link"

Where the link must be the *text* of the link. If you pass a CSS selector, it's not going to work. If I changed search_submit to be a CSS selector, it'll fail. Believe me, I've tried it a bunch of times.

Got it? Ok: in reality, the named selector lets you cheat a little bit. In addition to the true "text" of a field, it also searches for the name attribute and the id attribute. That's why our scenario works.

But *please please* - don't use the name or id. In fact, The cardinal rule in Behat is that you should never use CSS selectors or other technical things in your scenario. Why? Because the person who is benefiting from the feature is a web user, and we're writing this from their point of view. A web user doesn't understand what searchTerm or search_submit means. That makes your scenario less useful: it's technical jargon instead of behavior descriptions.

So why did we cheat? Well, the search field doesn't have a label and the button doesn't have any text. I can't use the named selector to find these, *unless* I cheat.

Custom Mink Definition

Whenever you want to cheat or can only find something via CSS, there's a simple solution: use new language and create a custom definition. Change the first line to:

18 lines | features/search.feature

```
... lines 1 - 9
10      When I fill in the search box with "<term>"
... lines 11 - 18
```

If I can't target it with real text, I'll just use some natural language. PhpStorm highlights the line because we don't have a definition function matching this text. For the second problem line, use

18 lines | features/search.feature

```
... lines 1 - 10
11      And I press the search button
... lines 12 - 18
```

You know the drill: it's time to run the scenario. It prints out the two functions we need to fill in:

102 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 85
    /**
     * @When I fill in the search box with :arg1
     */
    public function iFillInTheSearchBoxWith($arg1)
    {
        throw new PendingException();
    }

    /**
     * @When I press the search button
     */
    public function iPressTheSearchButton()
    {
        throw new PendingException();
    }
... lines 101 - 102
```

Getting the Mink Session

Filling these in shouldn't be hard: we're pretty good with Mink. But, how can we access the Mink Session? There's a couple ways to get it, but the easiest is to make FeatureContext extend RawMinkContext:

115 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 6
use Behat\MinkExtension\Context\RawMinkContext;

... lines 9 - 13
class FeatureContext extends RawMinkContext implements Context, SnippetAcceptingContext
... lines 15 - 115
```

This gives us access to a bunch of functions: the most important being getSession() and another called visitPath() that we'll use later:

166 lines | vendor/behat/mink-extension/src/Behat/MinkExtension/Context/RawMinkContext.php

```
... lines 1 - 105
    public function getSession($name = null)
    {
        return $this->getMink()->getSession($name);
    }

... lines 111 - 128
    public function visitPath($path, $sessionName = null)
    {
        $this->getSession($sessionName)->visit($this->locatePath($path));
    }
... lines 133 - 166
```

On the first method, change arg1 to term:

115 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 86
87      /**
88       * @When I fill in the search box with :term
89       */
90      public function iFillInTheSearchBoxWith($term)
... lines 91 - 115
```

Once you're inside of FeatureContext it's *totally* OK to use CSS selectors to get your work done.

Back in the browser, inspect the search box element. It doesn't have an id but it does have a name attribute - let's find it by that. Start with $searchBox = $this->getSession()->getPage(). Then, to drill down via CSS, add ->find('css', '[name="searchTerm"]');. I'm going to add an assertNotNull() in case the search box isn't found for some reason. Fill that in with $searchBox, 'The search box was not found':

115 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 91
92          $searchBox = $this->getSession()
93              ->getPage()
94              ->find('css', 'input[name="searchTerm"]');
95
96          assertNotNull($searchBox, 'Could not find the search box!');
... lines 97 - 115
```

Now that we have the individual element, we can take action on it with one of the cool functions that come with being an individual element, like attachFile, blur, check, click and doubleClick. One of them is setValue() that works for field. Set the value to $term.

115 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 97
98          $searchBox->setValue($term);
... lines 99 - 115
```

This is a perfect step definition: find an element and do something with it.

To press the search button, we can do the exact same thing.
$button = $this->getSession()->getPage()->find('css', '#search_submit');. And assertNotNull($button, 'The search button could not be found'). It's always a good idea to code defensively. This time, use the press() method:

115 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 103
104     public function iPressTheSearchButton()
105     {
106         $button = $this->getSession()
107             ->getPage()
108             ->find('css', '#search_submit');
109
110         assertNotNull($button, 'Could not find the search button!');
111
112         $button->press();
113     }
... lines 114 - 115
```

We're ready to run the scenario again. It passes!

That was more work, but it's a better solution. With no CSS inside of our scenarios, they're less dependent on the markup on our site and this is a heck of a lot easier to understand than before with the cryptic name and ids.

Create a getPage() Shortcut

To save time in the future, create a private function getPage() and return $this->getSession()->getPage();:

121 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 112
    /**
     * @return \Behat\Mink\Element\DocumentElement
     */
    private function getPage()
    {
        return $this->getSession()->getPage();
    }
... lines 120 - 121
```

I'll put a little PHPDoc above this so next month we'll remember what this is.

Now we can shorten both definition functions a bit with $this->getPage():

121 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 89
    public function iFillInTheSearchBoxWith($term)
    {
        $searchBox = $this->getPage()
            ->find('css', 'input[name="searchTerm"]');
... lines 94 - 97
    }
... lines 99 - 102
    public function iPressTheSearchButton()
    {
        $button = $this->getPage()
            ->find('css', '#search_submit');
... lines 107 - 110
    }
... lines 112 - 121
```

Test the final scenarios out. Perfect! Now we have access to Mink inside of FeatureContext *and* we know that including CSS inside of scenarios is not the best way to make friends.

The assertSession

One more quick shortcut. Thanks to the RawMinkContext base class, we also have access to a cool object called WebAssert through the assertSession() method. Replace getPage() with assertSession() and find() with elementExists(). Now, remove the assertNotNull() call:

117 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 13
class FeatureContext extends RawMinkContext implements Context, SnippetAcceptingContext
{
... lines 16 - 89
    public function iFillInTheSearchBoxWith($term)
    {
        $searchBox = $this->assertSession()
            ->elementExists('css', 'input[name="searchTerm"]');

        $searchBox->setValue($term);
    }
... lines 97 - 115
}
```

The elementExists *finds* the element *and* asserts that it exists all at once. Nice! Make the same changes for pressing the button:

117 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 100
    public function iPressTheSearchButton()
    {
        $button = $this->assertSession()
            ->elementExists('css', '#search_submit');

        $button->press();
    }
... lines 108 - 117
```

The WebAssert class has a bunch of other handy methods on it - check them out.

Chapter 11: Context Organization and Behat Suites

When Behat loads, it reads step definitions from FeatureContext and MinkContext because of the behat.yml setup:

12 lines | behat.yml

```
default:
    suites:
        default:
            contexts:
                - FeatureContext
                - Behat\MinkExtension\Context\MinkContext
... lines 7 - 12
```

This is a really powerful idea: instead of having one giant context class, we can break things down into as many small, organized pieces. We might have one context for dealing with adding users to the database and another for the API. If you look at *our* FeatureContext, we already have two very different ideas mixed together: some functions interact with the terminal and others help deal with a web page.

This is *begging* to be split into 2 classes. Let's copy FeatureContext and create a new file called CommandLineProcessContext. Update the class name and get rid of anything in here that doesn't help do things with the command line:

72 lines | features/bootstrap/CommandLineProcessContext.php

```
... lines 1 - 9
class CommandLineProcessContext implements Context, SnippetAcceptingContext
{
    private $output;
... lines 13 - 16
    public function iHaveAFileNamed($filename)
    {
        touch($filename);
    }
... lines 21 - 24
    public function iRun($command)
    {
        $this->output = shell_exec($command);
    }
... lines 29 - 32
    public function iShouldSeeInTheOutput($string)
    {
        assertContains(
            $string,
            $this->output,
            sprintf('Did not see "%s" in output "%s"', $string, $this->output)
        );
    }
... lines 41 - 44
    public function moveIntoTestDir()
    {
        if (!is_dir('test')) {
            mkdir('test');
        }
        chdir('test');
    }
... lines 52 - 55
    public function moveOutOfTestDir()
    {
        chdir('..');
        if (is_dir('test')) {
            system('rm -r '.realpath('test'));
        }
    }
... lines 63 - 66
    public function iHaveADirNamed($dir)
    {
        mkdir($dir);
    }
}
```

In FeatureContext, do the opposite: remove all the things that have nothing to do with working on a web site. Delete these functions and our before and after scenario hooks:

57 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 13
class FeatureContext extends RawMinkContext implements Context, SnippetAcceptingContext
{
... lines 16 - 22
    public function __construct()
    {
    }
... lines 26 - 29
    public function iFillInTheSearchBoxWith($term)
    {
        $searchBox = $this->assertSession()
            ->elementExists('css', 'input[name="searchTerm"]');

        $searchBox->setValue($term);
    }
... lines 37 - 40
    public function iPressTheSearchButton()
    {
        $button = $this->assertSession()
            ->elementExists('css', '#search_submit');

        $button->press();
    }
... lines 48 - 51
    private function getPage()
    {
        return $this->getSession()->getPage();
    }
}
```

That's a lot clearer.

Of course to keep our tests passing, we need to tell Behat about our new context:

13 lines | behat.yml

```
... lines 1 - 3
            contexts:
                - FeatureContext
                - CommandLineProcessContext
                - Behat\MinkExtension\Context\MinkContext
... lines 8 - 13
```

If we run behat now:

```
$ $ ./vendor/bin/behat
```

It should run all of our features: the ls *and* web stuff. It does, and it works! Ignore the undefined functions - those are from product_admin.feature: we haven't finished that yet.

Multiple Suites

But we can go further. in behat.yml, check out the suites key. Currently, we have one "suite" called default:

13 lines | behat.yml

```
default:
    suites:
        default:
... lines 4 - 13
```

But you could have many. What's a suite? It's a combination of a set of feature files and the contexts that should be used for them. Think about it: the ls.feature is the only feature that needs CommandLineProcessContext. And every other feature *only* needs FeatureContext and MinkContext. This is the perfect use-case for a second suite that I'm going to call commands. In this case, only add the CommandLineProcessContext:

18 lines | behat.yml

```
default:
    suites:
        default:
... lines 4 - 7
        commands:
            contexts:
                - CommandLineProcessContext
... lines 11 - 18
```

Remove that from the default suite:

18 lines | behat.yml

```
default:
    suites:
        default:
            contexts:
                - FeatureContext
                - Behat\MinkExtension\Context\MinkContext
... lines 7 - 18
```

When you execute Behat, it uses the default suite unless you tell it which one to use with the --suite option. Try it with --suite=commands and then run ls.feature:

```
$ $ ./vendor/bin/behat --suite=commands features/ls.feature
```

Or you can use the -dl option to see only the definition lists associated with the contexts in that suite:

```
$ $ ./vendor/bin/behat --suite=commands features/ls.feature -dl
```

Without --suite, we see definitions for the default suite:

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

And yes, we can go *even* further by telling each suites which features belong to them. Under the features/ directory, create two new directories called commands and web Let's organize: move ls.feature into commands/ and the other four features into web/. Now, add a paths key to the default suite and set it to [%paths.base%/features/web]:

18 lines | behat.yml

```
default:
    suites:
        default:
            contexts:
... lines 5 - 6
            paths: [ %paths.base%/features/web ]
... lines 8 - 18
```

%paths.base% is a shortcut to the root of the project. For the commands suite, do the same thing to point to commands/:

18 lines | behat.yml

```
default:
    suites:
        default:
... lines 4 - 7
        commands:
... lines 9 - 10
            paths: [ %paths.base%/features/commands ]
... lines 12 - 18
```

Now, if you run the default suite:

```
$ $ ./vendor/bin/behat
```

Behat knows to only execute the features in the web/ directory. With --suite=commands, it *only* runs the features inside of commands/:

```
$ $ ./vendor/bin/behat --suite=commands
```

So if you have two very different things that are being tested, consider separating them into different suites entirely. But at the very least, use multiple contexts to keep organized and stay sane.

Chapter 12: Building a Login Scenario

Open up authentication.feature, because, you can't do much until you login. Let's add a scenario:

```
13 lines | features/web/authentication.feature
... lines 1 - 5
6     Scenario: Logging in
... lines 7 - 13
```

We remember from running behat -dl:

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

That we have a lot of built in language already. Let's save some effort and describe the login process using these:

```
13 lines | features/web/authentication.feature
... lines 1 - 6
7       Given I am on "/"
... lines 8 - 13
```

We want to start on the homepage. PhpStorm tries to help by auto-completing this step, but when I hit tab, it prints it with extra stuff. When you use the -dl option, the "Given I am on" ends with a bunch of crazy regex. Anytime you see regex like this, it's just forming a wildcard: something where you can pass any value, surrounded by quotes.

Oh, and one other thing: even though each line starts with Given, When and Then in the definition list, that first word doesn't matter. We could actually say:

```
Then I am on "/"
```

It doesn't sound right in English, but technically, it would still run.

Alright! Given I am on "/". Next, I will click "Login". The built-in definition for that is "I follow". There's no built-in definition for "I click" but we'll add one later since that's how most people actually talk.

But for now let's add:

```
13 lines | features/web/authentication.feature
... lines 1 - 7
8       When I follow "Login"
... lines 9 - 13
```

Remember, these all use the named selector, so we're using "Login" because that's the text of the link.

On the login page, we need to fill in these two fields. Again, because of the named selector, we'll target these by the labels "Username" and "Password". There are a few definitions for fields, byt the one I like is "When I fill in field with value". So:

```
13 lines | features/web/authentication.feature
... lines 1 - 8
9       And I fill in "Username" with "admin"
10      And I fill in "Password" with "adminpass"
... lines 11 - 13
```

Yes I know, that isn't the right password - this won't work yet. It's cool though.

Finally, press the login button with:

```
13 lines | features/web/authentication.feature
... lines 1 - 10
11      And I press "Login"
... lines 12 - 13
```

Notice that you *follow* a link but you *press* a button. Then we need to find something to assert - some sign that this worked. Login on the browser. Hmm, nothing says "Congratulations, you're in!"... but our login button *did* change to "Logout". Let's look for that with:

```
13 lines | features/web/authentication.feature
... lines 1 - 11
12      Then I should see "Logout"
```

Good work team. Let's run this!

```
$ ./vendor/bin/behat features/web/authentication.feature
```

Debugging Failed Scenarios

It runs... and fails:

> The text "Logout" was not found anywhere in the text of the current page.

I *happen* to know this is because the password is wrong. But let's pretend that we *didn't* know that, and we're staring at this message wondering what the heck is going on.

Debugging tip number 1: right before the failing step, use a step definition called:

```
14 lines | features/web/authentication.feature
... lines 1 - 11
12      Then print last response
... lines 13 - 14
```

Now run this again. It still fails, but first it prints out the entire page's code. Yes, this *is* ugly - I have another debugging tip later. There are two important things: first, we're still on the login page for some reason. And second, if you scan down, you'll see the error message: "invalid credentials".

Tip

You can also *see* the failed page by using Then show last response:

```
14 lines | features/web/authentication.feature
... lines 1 - 5
6     Scenario: Logging in
... lines 7 - 10
11      And I press "Login"
12      Then show last response
... lines 13 - 14
```

You'll just need to configure the show_cmd in behat.yml. On OSX, I use show %s, but using firefox %s is common on other systems:

19 lines | behat.yml

```
default:
... lines 2 - 12
    extensions:
        Behat\MinkExtension:
... lines 15 - 17
            show_cmd: 'open %s'
```

You can even set the show_auto setting to true to automatically open a browser on failures.

Let's remove our debug line and update this to the correct password which is "admin". And now let's rerun it:

```
$ ./vendor/bin/behat features/web/authentication.feature
```

It's alive!

But we have a *big* problem: We're assuming that there will *always* be an admin user in the database with password admin. What if there isn't? What if the intern deleted that or your dropped your database locally? Ah! Everything would start failing.

We can do better.

Chapter 13: Controlling the Database

Eventually, you'll need to insert test data at the beginning of your scenarios. And that's when things get tricky. So, let's learn to do this right.

First: never assume that there is data in the database at the beginning of a scenario. Instead, you should put any data you need there intentionally with a Given.

Go to the top of this scenario and add:

14 lines | features/web/authentication.feature

```
... lines 1 - 6
7       Given there is an admin user "admin" with password "admin"
... lines 8 - 14
```

I'm inventing this language - it describes something that needs to be setup using natural language. Change the Given line below this to And for readability:

14 lines | features/web/authentication.feature

```
... lines 1 - 7
8       And I am on "/"
... lines 9 - 14
```

Run it again: it should print out the step definition for this new language.

```
$ ./vendor/bin/behat features/web/authentication.feature
```

And it does! Copy that and put it into our FeatureContext class. Change :arg1 to :username and :arg2 to :password. Update the arguments to match:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 41
42      /**
43       * @Given there is an admin user :username with password :password
44       */
45      public function thereIsAnAdminUserWithPassword($username, $password)
46      {
... lines 47 - 54
55      }
... lines 56 - 87
```

To fill this in, we want to insert a new user in the database with this username and password. If Symfony were loaded, that would be really easy: I'd create a User object, set some data then persist and flush it. So let's do that! Even if you're not using Symfony, you'll use the same basic process to bootstrap your application to get access to all of your normal useful objects and functions.

Bootstrapping Symfony in FeatureContext

We only need to boot Symfony once at the beginning of the test suite. Afterwards all of our scenarios will be able to access Symfony's container. Make a new public function bootstrapSymfony():

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 31
32      public static function bootstrapSymfony()
33      {
... lines 34 - 39
40      }
... lines 41 - 87
```

And inside, we'll do exactly what its name says.

We'll need a couple of require statements for autoload.php and the AppKernel.php class:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 33
34          require_once __DIR__.'/../../app/autoload.php';
35          require_once __DIR__.'/../../app/AppKernel.php';
... lines 36 - 87
```

Then, it's as easy as $kernel = new AppKernel();. Pass it the environment - test - and the debug value - true - so we can see errors. Finish with $kernel->boot();:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 36
37          $kernel = new AppKernel('test', true);
38          $kernel->boot();
... lines 39 - 87
```

Congrats - you just bootstrapped your Symfony app.

What we really want is access to the service container. To get that, create a new private static $container; property:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 15
16      private static $container;
... lines 17 - 87
```

Then in the method, set that with self::$container = $kernel->getContainer();:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 38
39          self::$container = $kernel->getContainer();
... lines 40 - 87
```

Now, as long as we call bootstrapSymfony() first, we'll have access to the container. Oh, and update the method to be a public static function:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 31
32      public static function bootstrapSymfony()
... lines 33 - 87
```

I'm making this all static because it allows us to have one container across all of our different scenarios. Because remember, each scenario gets its own context instances.

We could call this method manually, but that's not fancy! Remember the hook system? We used @BeforeScenario and @AfterScenario before, but there are other hooks, like @BeforeSuite. Let's use that!

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 28
    /**
     * @BeforeSuite
     */
    public static function bootstrapSymfony()
... lines 33 - 87
```

Behat will call this method one time, even if we're testing 10 features and 100 scenarios.

Saving a New User

Inside of thethereIsAnAdminUserWithPassword() step definition, let's go to work! I already have a User entity setup in the project, so we can say $user = new User(). Then set the username and the "plainPassword":

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 46
        $user = new \AppBundle\Entity\User();
        $user->setUsername($username);
        $user->setPlainPassword($password);
... lines 50 - 87
```

I have a Doctrine listener already setup that will encode the password automatically. Which is good: it's well-known that raptors can smell un-encoded passwords...

In this app, to make this an "admin" user, we need to give the user ROLE_ADMIN:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 49
        $user->setRoles(array('ROLE_ADMIN'));
... lines 51 - 87
```

Now the moment of truth: we need the entity manager. Cool! Grab it with $em = self::$container->get('doctrine') (to get the Doctrine service) and then ->getManager();:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 51
        $em = self::$container->get('doctrine')->getManager();
... lines 53 - 87
```

It's easy from here: $em->persist($user); and $em->flush();:

87 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 52
        $em->persist($user);
        $em->flush();
... lines 55 - 87
```

And that should do it!

We already have a user called admin in the database, so let's test this using admin2. Give it a go!

```
$ ./vendor/bin/behat features/web/authentication.feature
```

This should not work, since this user isn't in the database yet... unless it's created by the code we just wrote! Brilliant!

This is huge: we're guaranteeing there's an admin2 user by bootstrapping our app and being dangerous with all of our useful services.

Chapter 14: The SymfonyExtension & Clearing Data Between Scenarios

Change the user and pass back to match the original user in the database: "admin" and "admin":

14 lines | features/web/authentication.feature

```
... lines 1 - 6
7       Given there is an admin user "admin" with password "admin"
... lines 8 - 14
```

Now rerun the scenario:

```
$ ./vendor/bin/behat features/web/authentication.feature
```

Boom! This time it explodes!

> Integrity constraint violation: UNIQUE constraint failed: user.username

We already have a user called "admin" in the database... and since I made that a unique column, creating *another* user in Given is putting a stop to our party.

Clearing the Database Before each Scenario

Important point: you should start every scenario with a blank database. Well, that's not 100% true. What I want to say is: you should start every scenario with a predictable database. Some projects have look-up tables - like a "product status" table with rows for in stock, out of stock, back ordered, etc. I really hate these, but anyways, sometimes there are tables that *need* to be filled in for anything to work. You'll want to empty the database before each scenario... except for any lookup tables.

Since *we* don't have any of these pesky look-up guys, we can empty everything before every scenario. To do this, we'll of course, use hooks.

Create a new public function clearData():

97 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 13
14  class FeatureContext extends RawMinkContext implements Context, SnippetAcceptingContext
15  {
... lines 16 - 44
45      public function clearData()
46      {
... lines 47 - 49
50      }
... lines 51 - 95
96  }
```

Clearing data now is pretty easy, since we have access to the entity manager via self::container->get('doctrine')->getManager();:

97 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 46
47          $em = self::$container->get('doctrine')->getManager();
... lines 48 - 97
```

Now we can issue DELETE queries on the two entities that we care about so far: product and user. I'll use $em->createQuery('DELETE FROM AppBundle:Product')->execute();:

97 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 47
48          $em->createQuery('DELETE FROM AppBundle:Product')->execute();
... lines 49 - 97
```

Copy and paste that line and change "Product" to "User":

97 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 48
49          $em->createQuery('DELETE FROM AppBundle:User')->execute();
... lines 50 - 97
```

Oh and make sure that says "Product" and not "Products". Activate all of this with the @BeforeScenario annotation:

97 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 41
42      /**
43       * @BeforeScenario
44       */
45      public function clearData()
... lines 46 - 97
```

Try it all again:

```
$ ./vendor/bin/behat features/web/authentication.feature
```

Perfect! We can run this over and over because it's clearing out the data first.

The Symfony2Extension

And, surprise! There's an easier way to bootstrap Symfony and clear out the database. I always like taking the long way first so we can see how things work.

First, install a new library called behat/symfony2-extension with --dev so it goes into my require dev section:

```
$ composer require behat/symfony2-extension --dev
```

An extension in Behat is a plugin. We're already using the MinkExtension:

19 lines | behat.yml

```
1   default:
... lines 2 - 12
13      extensions:
14          Behat\MinkExtension:
15              base_url: http://localhost:8000
... lines 16 - 19
```

Activate the new plugin in behat.yml: Behat\Symfony2Extension::

20 lines | behat.yml

```
1   default:
... lines 2 - 12
13      extensions:
14          Behat\MinkExtension:
... lines 15 - 18
19          Behat\Symfony2Extension: ~
```

And as luck would have it, it doesn't need any configuration. It looks like we still need to wait for it to finish installing in the

terminal... there we go!

The most important thing the Symfony2 Extension gives you is, access to Symfony's container... but wait, we already have that? Well, this just makes it easier.

Remove the private static $container; property and the bootstrapSymfony() function. Instead of these, we'll use a PHP 5.4 trait called KernelDictionary:

84 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 13
14  class FeatureContext extends RawMinkContext implements Context, SnippetAcceptingContext
15  {
16      use \Behat\Symfony2Extension\Context\KernelDictionary;
... lines 17 - 82
83  }
```

This gives us two new functions, getKernel(), but more importantly getContainer():

56 lines | vendor/behat/symfony2-extension/src/Behat/Symfony2Extension/Context/KernelDictionary.php

```
... lines 1 - 21
22  trait KernelDictionary
23  {
... lines 24 - 40
41      public function getKernel()
42      {
43          return $this->kernel;
44      }
... lines 45 - 50
51      public function getContainer()
52      {
53          return $this->kernel->getContainer();
54      }
55  }
```

It takes care of all of the booting of the kernel stuff for us, and it even reboots the kernel between each scenario so they don't run into each other. That's important because remember, each scenario should be completely independent of the others.

Search for the old self::$container code. Change it to $this->getContainer():

84 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 31
32      public function clearData()
33      {
34          $em = $this->getContainer()->get('doctrine')->getManager();
... lines 35 - 36
37      }
... lines 38 - 41
42      public function thereIsAnAdminUserWithPassword($username, $password)
43      {
... lines 44 - 48
49          $em = $this->getContainer()->get('doctrine')->getManager();
... lines 50 - 51
52      }
... lines 53 - 84
```

You see that PhpStorm all of a sudden auto-completes the methods on the services we fetch because it recognizes this as the container and so knows that this returns the entity manager.

Let's try things again!

```
$ ./vendor/bin/behat features/web/authentication.feature
```

Still works! But now with less effort. If you have multiple context classes, you can use the KernelDictionary on all of them to get access to the container.

Clearing the Database Easily

OK, so what about clearing the database? It'll be a huge pain to add more and more manual queries. Fortunately Doctrine gives us a better way: a Purger. Create a new variable called $purger and set it to a new ORMPurger(). Pass it the entity manager:

84 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 32
33      public function clearData()
34      {
35          $purger = new ORMPurger($this->getContainer()->get('doctrine')->getManager());
... line 36
37      }
... lines 38 - 84
```

After that, type $purger->purge();, and that's it:

84 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 35
36          $purger->purge();
... lines 37 - 84
```

This will go through each entity and clear out all of your data. If it's working, then our tests should pass:

```
$ ./vendor/bin/behat features/web/authentication.feature
```

And they do! Same functionality and a lot less code. For bigger databases with lots of lookup tables, it may be too much to clear every table and re-add all the data you need. In those cases, trying experimenting with creating a SQL file that populates the database and executing that before each scenario. Or, populate an SQLite file with whatever you want to start with, then copy this and use it as your database before each test. That's a super-fast way to roll back to your known data set.

Chapter 15: Practice: Find Elements, Login with 1 Step and Debug

Look at the Product Admin Feature. When we built this earlier, we were planning the feature and learning how to write really nice scenarios. *Now* we know that most of the language we used matches the built-in definitions that Mink gives us for free.

Time to make these pass! Run *just* the "List available products" scenario on line 6. To do that type, ./vendor/bin/behat point to the file and then add :6:

```
$ ./vendor/bin/behat features/product_admin.feature:6
```

The number 6 is the line where the scenario starts. This prints out some missing step definitions, so go head and copy and paste them into the FeatureContext class:

129 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 76
77      /**
78       * @Given there are :count products
79       */
80      public function thereAreProducts($count)
81      {
... lines 82 - 91
92      }
... line 93
94      /**
95       * @When I click :linkName
96       */
97      public function iClick($linkName)
98      {
... line 99
100     }
101
102     /**
103      * @Then I should see :count products
104      */
105     public function iShouldSeeProducts($count)
106     {
... lines 107 - 110
111     }
... lines 112 - 129
```

Say what you need, but not more

For the thereAreProducts() function, change the variable to count and create a for loop:

129 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 76
77      /**
78       * @Given there are :count products
79       */
80      public function thereAreProducts($count)
81      {
82          for ($i = 0; $i < $count; $i++) {
... lines 83 - 88
89          }
... lines 90 - 91
92      }
... lines 93 - 129
```

Inside, create some products and put some dummy data on each one:

129 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 81
82          for ($i = 0; $i < $count; $i++) {
83              $product = new Product();
84              $product->setName('Product '.$i);
85              $product->setPrice(rand(10, 1000));
86              $product->setDescription('lorem');
... lines 87 - 88
89          }
... lines 90 - 129
```

Why dummy data? The definition says that we need 5 products: but it doesn't say what those products are called or how much they cost, because we don't care about that for this scenario. The point is: only include details in your scenario that you actually care about.

We'll need the entity manager in a lot of places, so create a private function getEntityManager() and return $this->getContainer()->get() and pass it the service name that points directly to the entity manager:

129 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 120
121     /**
122      * @return \Doctrine\ORM\EntityManager
123      */
124     private function getEntityManager()
125     {
126         return $this->getContainer()->get('doctrine.orm.entity_manager');
127     }
... lines 128 - 129
```

Perfect!

Back up in thereAreProducts(), add $em = $this->getEntityManager(); and the usual $em->persist($product); and an $em->flush(); at the bottom. This is easy stuff now that we've got Symfony booted:

```
129 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 81
82          for ($i = 0; $i < $count; $i++) {
    ... lines 83 - 87
88              $this->getEntityManager()->persist($product);
89          }
90
91          $this->getEntityManager()->flush();
    ... lines 92 - 129
```

Using "I Click" to be more Natural

Go to the next method - iClick() - and update the argument to $linkText:

```
129 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 93
94      /**
95       * @When I click :linkName
96       */
97      public function iClick($linkName)
    ... lines 98 - 129
```

We want this to work just like the built-in "I follow" function. In fact, the only reason we're not just re-using that language is that nobody talks like that: we click things.

Anyways, the built-in functionality finds the link by its text, not a CSS selector. To use the named selector, add $this->getPage()->findLink(), pass it $linkText and then call click(); on that. Oh heck, let's be even lazier: just say, ->clickLink(); and be done with it:

```
129 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 98
99          $this->getPage()->clickLink($linkName);
    ... lines 100 - 129
```

This looks for a link inside of page and then clicks it.

Finally, in iShouldSeeProducts(), we're asserting that a certain number of products are shown on the page:

```
129 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 101
102     /**
103      * @Then I should see :count products
104      */
105     public function iShouldSeeProducts($count)
    ... lines 106 - 129
```

In other words, once we get into the Admin section, we're looking for the number of rows in the product table.

There aren't any special classes to help find *this* table, but there's only one on the page, so find it via the table class:

```
129 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 106
107         $table = $this->getPage()->find('css', 'table.table');
    ... lines 108 - 129
```

Next, use assertNotNull() in case it doesn't exist:

```
129 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 107
108         assertNotNull($table, 'Cannot find a table!');
    ... lines 109 - 129
```

Now, use assertCount() and pass it intval($count) as the first argument:

```
129 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 109
110         assertCount(intval($count), $table->findAll('css', 'tbody tr'));
    ... lines 111 - 129
```

For the second argument, we need to find *all* of the <tr> elements inside of the table's <tbody>. Remember, once you find an element, you can search *inside* of it with find() or $table->findAll() to return an array of elements instead of just one. And don't forget that the first argument is still css: PhpStorm is yelling at me because I like to forget this. Ok, let's try that out!

```
$ ./vendor/bin/behat features/product_admin.feature:6
```

Debugging Failures!

Ok, it gets *further* but still fails. It says:

> Link "Products" not found

It's trying to find a link with the word "Products" but isn't having much luck. I wonder why? We need to debug! Right before the error, add:

```
And print last response
```

Run that one again:

```
$ ./vendor/bin/behat features/product_admin.feature:6
```

Scroll up... up... up... all the way up to the top. Ahhh of course! We're on the login page. We forgot to login, so we're getting kicked back here.

Logging in... in one Step!

We already did all that login stuff in authentication.feature, and I'm tempted to copy and paste all of those lines to the top of this scenario:

```
14 lines | features/web/authentication.feature
    ... lines 1 - 7
8       And I am on "/"
9       When I follow "Login"
10      And I fill in "Username" with "admin"
11      And I fill in "Password" with "admin"
12      And I press "Login"
    ... lines 13 - 14
```

But, it would be pretty lame to need to put *all* of this at the top of pretty much every scenario. You know what would be cooler? To just say:

```
21 lines | features/web/product_admin.feature
    ... lines 1 - 6
7       Given I am logged in as an admin
    ... lines 8 - 21
```

Ooo another new step definition will be needed! Rerun the test and copy the function that behat so thoughtfully provides for us. As usual, put this in FeatureContext:

142 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 112
113     /**
114      * @Given I am logged in as an admin
115      */
116     public function iAmLoggedInAsAnAdmin()
117     {
... lines 118 - 123
124     }
... lines 125 - 142
```

Using Mink, we'll do *all* the steps needed to login. First, go to the login page. Normally you'd say $this->getSession()->visit('/login'). But don't! Instead, wrap /login in a call to $this->visitPath():

142 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 115
116     public function iAmLoggedInAsAnAdmin()
117     {
... lines 118 - 119
120         $this->visitPath('/login');
... lines 121 - 123
124     }
... lines 125 - 142
```

This prefixes /login - which isn't a full URL - with our base URL: http://localhost:8000.

Once we're on the login page, we need to fill out the username and password fields and press the button. We could find this stuff with CSS, but the named selector is a lot easier. Say $this->getPage()->findField('Username')->setValue(). Ah, let's be lazier and do this all at once with fillField(). Pass this the label for the field - Username - and the value to fill in:

142 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 115
116     public function iAmLoggedInAsAnAdmin()
117     {
... lines 118 - 119
120         $this->visitPath('/login');
121         $this->getPage()->fillField('Username', 'admin');
... lines 122 - 123
124     }
... lines 125 - 142
```

But hold on: before we fill in the rest, don't we need to make sure that this user exists in the database? Absolutely, and fortunately, we already have a function that creates a user: thereIsAnAdminUserWithPassword(). Call that from our function and pass it the usual admin / admin:

142 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 115
116     public function iAmLoggedInAsAnAdmin()
117     {
118         $this->thereIsAUserWithPassword('admin', 'admin');
119
120         $this->visitPath('/login');
... lines 121 - 123
124     }
... lines 125 - 142
```

Finish by filling in the password field and pressing the button. For that, there's another shortcut: instead of findButton() then

press(), use pressButton('Login'):

142 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 115
116     public function iAmLoggedInAsAnAdmin()
117     {
... lines 118 - 120
121         $this->getPage()->fillField('Username', 'admin');
122         $this->getPage()->fillField('Password', 'admin');
123         $this->getPage()->pressButton('Login');
124     }
... lines 125 - 142
```

This reproduces the steps from the login scenario, so that should be it! Run it!

```
$ ./vendor/bin/behat features/product_admin.feature:6
```

We're in great shape.

Chapter 16: When *I* do Something: Handling the Current User

Time for a challenge! Whenever you have products in the admin area, it either shows the name of the user that created it - like admin - or anonymous if it was created via some other method without an author. Right now, our admin area lists "anonymous" next to every product. The reason is simple: we're not setting the author when we create the products in FeatureContext.

I want to test that this table *does* show the correct author when its set. Create a new scenario to describe this:

28 lines | features/web/product_admin.feature

```
... lines 1 - 12
13    Scenario: Products show owner
14      Given I am logged in as an admin
... lines 15 - 28
```

Instead of just saying there are five products I'll say:

28 lines | features/web/product_admin.feature

```
... lines 1 - 14
15      And I author 5 products
... lines 16 - 28
```

This *is* new language that will need a step definition. To save time, we can go directly to the products page:

28 lines | features/web/product_admin.feature

```
... lines 1 - 15
16      When I go to "/admin/products"
17      # no products will be anonymous
18      Then I should not see "Anonymous"
... lines 19 - 28
```

Since "I" - some admin user - will be the author of the products, they should all show "admin": none will say "Anonymous". And we will *only* have these 5 products because we're clearing the database between each scenario to keep things independent.

Run just this new scenario by using its line number:

```
$ ./vendor/bin/behat features/web/product_admin.feature:13
```

Great - copy the iAuthorProducts() function code and paste it into our handy FeatureContext class - near the other product function:

160 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 85
86      /**
87       * @Given I author :count products
88       */
89      public function iAuthorProducts($count)
90      {
... line 91
92      }
... lines 93 - 160
```

These two functions will be similar, so we should reuse the logic. Copy the internals of thereAreProducts, make a new private function createProducts(). Pass it $count as an argument and also an optional User object which will be the author for those products:

```
160 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 141
142     private function createProducts($count, User $author = null)
143     {
144         for ($i = 0; $i < $count; $i++) {
145             $product = new Product();
146             $product->setName('Product '.$i);
147             $product->setPrice(rand(10, 1000));
148             $product->setDescription('lorem');
    ... lines 149 - 153
154             $this->getEntityManager()->persist($product);
155         }
156
157         $this->getEntityManager()->flush();
158     }
    ... lines 159 - 160
```

Now, add an if statement that says, if $author is passed then, $product->setAuthor():

```
160 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 149
150             if ($author) {
151                 $product->setAuthor($author);
152             }
    ... lines 153 - 160
```

I already have that relationship setup with in Doctrine. Great!

In thereAreProducts(), change the body of this function to $this->createProducts($count);:

```
160 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 80
81     public function thereAreProducts($count)
82     {
83         $this->createProducts($count);
84     }
    ... lines 85 - 160
```

Do the same thing in iAuthorProducts() for now:

```
160 lines | features/bootstrap/FeatureContext.php
    ... lines 1 - 88
89     public function iAuthorProducts($count)
90     {
91         $this->createProducts($count);
92     }
    ... lines 93 - 160
```

Clearly, this is still not setting the author. But I want to see if it executes first and then we'll worry about setting the author.

Who is "I" in a Scenario?

Cool! It runs... and fails because anonymous *is* still shown on the page. The question now is: how do we get the current user? The step says "I author". But who is "I" in this case? In product_admin.feature:

28 lines | features/web/product_admin.feature

```
... lines 1 - 5
6     Scenario: List available products
7       Given I am logged in as an admin
... lines 8 - 11
12
13    Scenario: Products show owner
14      Given I am logged in as an admin
... lines 15 - 28
```

You can see that "I" is whomever we logged in as. We didn't specify what the username should be for that user, but whoever is logged in is who "I" represents.

When we worked with the ls scenarios earlier, we needed to share the command output string between the steps of a scenario. In this case, we have a similar need: we need to share the user object from the step where we log in, with the step where "I" author some products. To share data between steps, create a new private $currentUser;:

164 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 16
17    class FeatureContext extends RawMinkContext implements Context, SnippetAcceptingContext
18    {
... lines 19 - 20
21        private $currentUser;
... lines 22 - 162
163   }
```

In iAmLoggedInAsAnAdmin(), add $this->currentUser = $this->thereIsAnAdminUserWithPassword():

164 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 119
120       public function iAmLoggedInAsAnAdmin()
121       {
122           $this->currentUser = $this->thereIsAUserWithPassword('admin', 'admin');
... lines 123 - 127
128       }
... lines 129 - 164
```

Click to open that function. It creates the User object of course, but now we need to also make sure it returns that:

164 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 42
43        /**
44         * @Given there is an admin user :username with password :password
45         */
46        public function thereIsAnAdminUserWithPassword($username, $password)
47        {
... lines 48 - 56
57            return $user;
58        }
... lines 59 - 164
```

And that's it! This login step will cause the currentUser property to be set and in iAuthorProducts() we can access that and pass it into createProducts() so that each product us authored by us:

164 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 89
90      /**
91       * @Given I author :count products
92       */
93      public function iAuthorProducts($count)
94      {
95          $this->createProducts($count, $this->currentUser);
96      }
... lines 97 - 164
```

It's pretty common to want to know *who* is logged in, so you'll likely want to use this in your project.

And hey it even passes! Now you can continue to write scenarios in terms of actions that "I" take and we will actually know who "I" is.

Chapter 17: Practicing BDD: Plan, then Build

One of the most beautiful things about Behat is having the opportunity to do behavior driven development. This is when you write the feature first, the scenarios second and *then* you code it. You *plan* the behavior, and *then* make it come to life.

So far... we haven't really been doing that. We have a existing site and we've been writing features and scenarios to describe how it *already* behaves. That's ok, and sometimes you'll do that in your real development life. But now it's time to do BDD correctly.

Step 1: Describe the Scenario

In the "Add a new product" scenario we're describing a feature that does *not* exist on the site yet. We planned this scenario earlier by planning - basically imagining - what the best behavior should be. And oops, I just noticed a typo. This should be:

```
29 lines | features/web/product_admin.feature
... lines 1 - 22
23     When I click "New Product"
... lines 24 - 29
```

Step 2: Execute Behat

With that fixed, let's start to bring this feature to life by running *just* this scenario:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

Our mission is clear: code *just* enough to fix a failure, then re-execute Behat and repeat until it's all green. The first failure is from "When I click 'New Product'". That makes sense: that link doesn't exist. But there is something else going on too. Add an "And print last response" line and try again:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

Of course: we also haven't logged in yet. Copy that line from the other scenario:

```
29 lines | features/web/product_admin.feature
... lines 1 - 20
21     Given I am logged in as an admin
22     And I am on "/admin/products"
... lines 23 - 29
```

We didn't think of this during the design phase, but clearly we need to be logged in. Try things again:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

Step 3: Add (a little) Code to make the Step Pass

Same failure, but now for the right reason: we're missing that link. Time to add it.

Open up the template for this page - list.html.twig. A link would look real nice up top. Don't add the href yet: just put in the text "New Product" and make it look nice with some CSS classes and a little icon:

67 lines | app/Resources/views/product/list.html.twig

```
... lines 1 - 12
            <a href="" class="btn btn-primary pull-right">
                <span class="fa fa-plus"></span> New Product
            </a>
... lines 16 - 67
```

Other than some easy-win styling, I want to do as little work as possible to get each step of the scenario to pass.

Refresh: there's the button. It doesn't go anywhere yet, try it:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

Step 4: Repeat until Green!

This time, it *did* click "New Product", but fails because it doesn't see any fields called "Name". No surprise: the link doesn't have an href and we don't even have a new products page.

To get this step to pass, I guess we need to create that. In ProductAdminController, make a new public function newAction() and set its URL to /admin/products/new. Name the route product_new so we can link to it. Inside the method, render a template called product/new.html.twig:

35 lines | src/AppBundle/Controller/ProductAdminController.php

```
... lines 1 - 26
    /**
     * @Route("/admin/products/new", name="product_new")
     */
    public function newAction()
    {
        return $this->render('product/new.html.twig');
    }
... lines 34 - 35
```

Easy enough!

In the product/ directory create that template: new.html.twig. Extend the base layout - layout.html.twig - and add add a body block. Add a form tag and make it submit right back to this same URL with method="POST":

25 lines | app/Resources/views/product/new.html.twig

```
{% extends 'layout.html.twig' %}

{% block body %}
    <form action="{{ path('product_new') }}" method="POST">
... lines 5 - 22
    </form>
{% endblock %}
```

I am not going to use Symfony's form system for this because (a) we don't have to and (b) my only goal is to get these tests passing. If you *did* want to use Symfony's form system, this is when you would start doing that!

To keep this moving, I'll paste in a bunch of code that creates the three fields we're referencing in the scenario: Name, Price and Description:

25 lines | app/Resources/views/product/new.html.twig

```
... lines 1 - 4
        <fieldset>
            <legend>New Product</legend>

            <div class="form-group">
                <label for="article-name">Name</label>
                <input type="text" class="form-control" id="article-name" name="name" />
            </div>
            <div class="form-group">
                <label for="article-price">Price</label>
                <input type="text" class="form-control" id="article-price" name="price" />
            </div>
            <div class="form-group">
                <label for="article-body">Description</label>
                <textarea class="form-control" id="article-body" name="description"></textarea>
            </div>
        </fieldset>

        <button type="submit" class="btn btn-primary">Save</button>
... lines 23 - 25
```

The important thing is that those names match up with the label text and that each label has a for attribute that points to the id of its field. This is how Mink can find the label by text and then find its field.

At the bottom, we have a save button that matches the second to last step:

29 lines | features/web/product_admin.feature

```
... lines 1 - 19
    Scenario: Add a new product
... lines 21 - 26
      And I press "Save"
... lines 28 - 29
```

Ok, try running the scenario again!

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

It's still failing in the same spot. This *might* seem weird, but if you debug this, you'll see that I forgot to fill in the "New Products" href:

25 lines | app/Resources/views/product/new.html.twig

```
... lines 1 - 3
    <form action="{{ path('product_new') }}" method="POST">
... lines 5 - 22
    </form>
... lines 24 - 25
```

My bad!

Run Behat again:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

We got further - it filled out and submitted the form, but didn't see the "Product Created FTW!" flash message. Time to add form processing logic.

Add Symfony's Request object as an argument with a type hint. Inside of newAction() add a simple

if ($request->isMethod('POST')):

```
41 lines | src/AppBundle/Controller/ProductAdminController.php
    ... lines 1 - 29
30      public function newAction(Request $request)
31      {
32          if ($request->isMethod('POST')) {
    ... lines 33 - 35
36          }
    ... lines 37 - 38
39      }
    ... lines 40 - 41
```

To be super lazy, what if we cheated by *not* saving the product and *only* showing that flash message? The site already has some flash messaging functionality, so add the message that the step is looking for: $this->addFlash('success', 'Product created FTW!'). Finish by redirecting the user to the product page:

```
41 lines | src/AppBundle/Controller/ProductAdminController.php
    ... lines 1 - 31
32          if ($request->isMethod('POST')) {
33              $this->addFlash('success', 'Product created FTW!');
34
35              return $this->redirectToRoute('product_list');
36          }
    ... lines 37 - 41
```

Run Behat:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

BDD for Fixing Bugs

It passes, even though the product isn't saved. Ok, don't be a big jerk and make your tests purposefully pass like this without coding the real functionality. But sometimes, you'll write a scenario and discover later that there's a bug because you forgot to test one part of the behavior. In this case I would improve my scenario before fixing the bug by adding:

```
30 lines | features/web/product_admin.feature
    ... lines 1 - 19
20  Scenario: Add a new product
    ... lines 21 - 28
29      And I should see "Veloci-chew toy"
```

With this, the scenario won't pass *unless* the product actually shows up on the product list. This is BDD: add a step to the scenario, watch it fail, and *then* code until it passes.

To fix this failure, add $product = new Product(); with code to set the name of the product. Copy that and repeat for price and description:

50 lines | src/AppBundle/Controller/ProductAdminController.php

```
... lines 1 - 31
32          if ($request->isMethod('POST')) {
... lines 33 - 34
35              $product = new Product();
36              $product->setName($request->get('name'));
37              $product->setDescription($request->get('description'));
38              $product->setPrice($request->get('price'));
... lines 39 - 44
45          }
... lines 46 - 50
```

This is missing validation, so you would do more work than this in real life, *maybe* with a scenario that guarantees that validation works.

Finish this with $em = $this->getDoctrine()->getManager();, then persist and flush:

50 lines | src/AppBundle/Controller/ProductAdminController.php

```
... lines 1 - 31
32          if ($request->isMethod('POST')) {
... lines 33 - 39
40              $em = $this->getDoctrine()->getManager();
41              $em->persist($product);
42              $em->flush();
... lines 43 - 44
45          }
... lines 46 - 50
```

Bug fixed! Try it:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

It's green! In fact, we can go to the browser, refresh and see the Veloci-chew toy for $20.

But there's a problem: it says that the author is "anonymous". This should be "admin" since I created it under that user. That's definitely a bug: we forgot to set the author in ProductAdminController. Ok, we know how to fix bugs using BDD: add a step to prove the bug:

32 lines | features/web/product_admin.feature

```
... lines 1 - 19
20    Scenario: Add a new product
... lines 21 - 30
31      And I should not see "Anonymous"
```

This is safe because there should only be the *1* product in the list. Back over to the terminal and run the test:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

Yes! The new step fails, phew!

In ProductAdminController, set the author when a product is created: $product->setAuthor($this->getUser());:

51 lines | src/AppBundle/Controller/ProductAdminController.php

```
... lines 1 - 31
32          if ($request->isMethod('POST')) {
... lines 33 - 34
35              $product = new Product();
... lines 36 - 38
39              $product->setAuthor($this->getUser());
... lines 40 - 45
46          }
... lines 47 - 51
```

Run Behat again:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

Fixed!

And that folks, is behavior driven development. It's useful, and a bucket of fun. It forces you to design the behavior of your code. But it also helps you know when you're finished. If the scenario is green, stop coding and over-perfecting things. And yes, *someone* will need to add a designer's touch for a really nice UI, But from a behavioral perspective, this feature does what it needs to do. So move onto what's next.

Chapter 18: Master JavaScript with Waits & Debugging

Let's talk JavaScript and the complications it can cause. Right now everything runs using the Goutte driver and background cURL requests. This means that if your behavior relies on JavaScript, it'll fail. We need to use Selenium - or Zombie.js - another supported JavaScript driver - for these scenarios.

I want this "New Product" button to *not* load a whole new page, but instead open up a modal. I cheated and already did most of the work for this. Inside of list.html.twig, add the class js-add-new-product to the "New Product" link:

67 lines | app/Resources/views/product/list.html.twig

```
... lines 1 - 12
            <a href="{{ path('product_new') }}" class="btn btn-primary pull-right js-add-new-product">
                <span class="fa fa-plus"></span> New Product
            </a>
... lines 16 - 67
```

This triggers some JavaScript that I have at the bottom of the template:

67 lines | app/Resources/views/product/list.html.twig

```
... lines 1 - 45
{% block javascripts %}
    {{ parent() }}

    <script type="text/javascript">
        $(document).ready(function() {
            $('.js-add-new-product').on('click', function(e) {
                e.preventDefault();

                var $modalContentHolder = $('#modal-content-holder');

                jQuery.ajax({
                    'url': $(this).attr('href'),
                    'success': function(content) {
                        $modalContentHolder.find('.modal-body').html(content);
                        $modalContentHolder.modal();
                    }
                });
            });
        });
    </script>
{% endblock %}
```

Next, make the new.html.twig template only return a partial page by removing the extends and Twig block tags:

21 lines | app/Resources/views/product/new.html.twig

```
<form action="{{ path('product_new') }}" method="POST">
    <fieldset>
        <legend>New Product</legend>

        <div class="form-group">
            <label for="article-name">Name</label>
            <input type="text" class="form-control" id="article-name" name="name" />
        </div>
        <div class="form-group">
            <label for="article-price">Price</label>
            <input type="text" class="form-control" id="article-price" name="price" />
        </div>
        <div class="form-group">
            <label for="article-body">Description</label>
            <textarea class="form-control" id="article-body" name="description"></textarea>
        </div>
    </fieldset>

    <button type="submit" class="btn btn-primary">Save</button>
</form>
```

This will now only be loaded via AJAX. There are cooler ways to make this all work, but for the purposes of Behat, they all face the same complications that you'll see.

Now click the "New Product" button. It opens up in a modal and it even saves my $34 Foo product. Simple!

Since we just modified our code, we should rerun our tests to make sure that everything still works. Run the new product scenario:

```
$ ./vendor/bin/behat features/product_admin.feature:19
```

It works! Wait... it shouldn't! It relies on JavaScript! In reality, the test clicked to this URL, went to an ugly, but functional page, filled out the form and hit "Save". That's kind of ok, it *did* test the form's functionality. But in reality, clicking the link opens a modal, so that should happen in the test too. Add @javascript to the top of the scenario:

33 lines | features/web/product_admin.feature

```
... lines 1 - 19
    @javascript
    Scenario: Add a new product
... lines 22 - 33
```

That changed bumped our scenario down one line. Put the new line into the terminal and run it:

```
$ ./vendor/bin/behat features/product_admin.feature:20
```

Just make sure that the Selenium server is still running in the background. Watch closely. Well, don't watch that silly Firefox error. We log in, go to the products page, clicks the "New Product" button, fill in the form fields, and hit "Save". It's perfect.

Waiting for things to Happen

Reality check: if I re-ran this 5 times, I bet it would fail at least once. Let me show you why. In ProductAdminController pretend this isn't such a fast ajax request. Add a sleep(1) to fake the time it would take to do some logic.

Re-run things:

```
$ ./vendor/bin/behat features/product_admin.feature:20
```

We log in, click the button... and the browser closes! We didn't see it fill out any fields. In the terminal, it failed at:

```
34 lines | features/web/product_admin.feature
... lines 1 - 20
21    Scenario: Add a new product
... lines 22 - 25
26      And I fill in "Name" with "Veloci-chew toy"
... lines 27 - 34
```

Because the "Name" field wasn't found. What is this madness?

Here's the secret: if you click a link or submit a form and it causes a full page refresh, Mink and Selenium will wait for that page refresh. But, if you do something in JavaScript, Selenium does *not* wait. It clicks the "New Products" link and *immediately* looks for the "Name" field. If that field isn't there almost instantly, it fails. We have to make Selenium wait after clicking the link.

To make this happen, add a new step like:

```
34 lines | features/web/product_admin.feature
... lines 1 - 20
21    Scenario: Add a new product
... lines 22 - 24
25      And I wait for the modal to load
... lines 26 - 34
```

Run Behat so it'll generate the missing definition. Selenium pops open the browser, then fails. Copy the new definition into FeatureContext:

```
175 lines | features/bootstrap/FeatureContext.php
... lines 1 - 129
130     /**
131      * @When I wait for the modal to load
132      */
133     public function iWaitForTheModalToLoad()
134     {
... lines 135 - 138
139     }
... lines 140 - 175
```

Yes!

Waiting the Wrong Way

Now, how do we wait for things? Well, there is a right way and a wrong way. Wrong way first! Add $this->getSession()->wait(5000); to wait for 5 seconds:

```
172 lines | features/bootstrap/FeatureContext.php
... lines 1 - 132
133     public function iWaitForTheModalToLoad()
134     {
135         $this->getSession()->wait(5000);
136     }
... lines 137 - 172
```

That should be overkill since the controller sleeps for just 1 second. Try this out anyways to see if it passes:

```
$ ./vendor/bin/behat features/product_admin.feature:20
```

The test logs us in, clicks the button 1...2...3...4...5, then it finally fills in the fields. It passed, but took too long. If you litter your test suite with wait statements like this, your tests will start to crawl. And you know what? You'll just stop running them, the fences will go down and guests will get eaten by dinosaurs in your park. Do you want your guests to be eaten? No. I didn't

think so. So let's look at the right way to do this.

Waiting the Right Way

The second argument to wait() is a JavaScript expression that will run on your page every 100 milliseconds. As soon as it equates to true, Mink will stop waiting and move onto the next step. I'm using Bootstrap's modal, and when it opens, an element with the class modal becomes visible. In your browser's console, try running $('.modal:visible').length. Because the modal is open, that returns one. Now close it: it returns zero. Pass this as the second argument to wait():

175 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 134
135         $this->getSession()->wait(
136             5000,
137             "$('.modal:visible').length > 0"
138         );
... lines 139 - 175
```

This now says: "Wait until this JavaScript expression is true or wait a maximum of 5 seconds." Why am I allowed to use jQuery here? Because this runs on your page and you have access to any JavaScript loaded by your app.

Run it again:

```
$ ./vendor/bin/behat features/product_admin.feature:20
```

This time it starts filling out the fields a lot faster. The most important thing for testing in JavaScript is mastering proper waits. I see people mess this up by using the "bad way" all the time.

Chapter 19: Debugging and Taking Screenshots with JavaScript

What about debugging in JavaScript? Change your scenario so it fails: change the "Name" field to look for a non-existent "Product Name":

34 lines | features/web/product_admin.feature

```
... lines 1 - 20
21    Scenario: Add a new product
... lines 22 - 25
26      And I fill in "Product Name" with "Veloci-chew toy"
... lines 27 - 34
```

Run it:

```
$ ./vendor/bin/behat features/product_admin.feature:20
```

Here's the problem: you can try to watch the browser, but it happens so quickly that it's hard to see what went wrong. In the terminal, the error tells us that there isn't a field called "Product Name", but with nothing else to help. Was there an error on the page? Are we on the wrong page? Is the field calling something else? Why won't someone tell us what's going on!?

Let me show you the master debugging tool. Google for behatch contexts. This is an open source library that has a bunch of useful contexts - classes like FeatureContext and MinkContext with free definitions. For example, this has a BrowserContext you could bring into your project to gain a bunch of useful definitions.

Pausing Selenium

I don't use this library directly, but I do steal from it. The DebugContext class has one of my favorite definitions: iPutABreakPoint(). Copy that and drop it into our FeatureContext file:

188 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 140
141     /**
142      * Pauses the scenario until the user presses a key. Useful when debugging a scenario.
143      *
144      * @Then (I )break
145      */
146     public function iPutABreakpoint()
147     {
148         fwrite(STDOUT, "\033[s    \033[93m[Breakpoint] Press \033[1;93m[RETURN]\033[0;93m to continue...\033[0m");
149         while (fgets(STDIN, 1024) == '') {}
150         fwrite(STDOUT, "\033[u");
151         return;
152     }
... lines 153 - 188
```

Or you could even create your own DebugContext if you wanted to organize things a bit. Shorten this to "I break". To use this, add this language directly *above* the "Product Name" step that's failing:

35 lines | features/web/product_admin.feature

```
... lines 1 - 20
21    Scenario: Add a new product
... lines 22 - 25
26      And break
27      And I fill in "Product Name" with "Veloci-chew toy"
... lines 28 - 35
```

The "I" part of this language is optional. Head back to the terminal to try this:

```
$ ./vendor/bin/behat features/product_admin.feature:20
```

This time, the modal pops open, and the browser freezes. The terminal just says: "Press [RETURN] to continue..." That's right: it's *waiting* for us to look at the page and debug the issue. Once we know what the problem is, hit enter to let it finish. This is my *favorite* way to debug!

Taking Screenshots

But there are more cool things, like iSaveAScreenshotIn(). Copy that definition and paste it into FeatureContext. Change the language to "I save a screenshot to" and remove $this->screenshotDir thing since we don't have that. To save screenshots to the root of your project, replace it with __DIR__'/../../':

199 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 153
154     /**
155      * Saving a screenshot
156      *
157      * @When I save a screenshot to :filename
158      */
159     public function iSaveAScreenshotIn($filename)
160     {
161         sleep(1);
162         $this->saveScreenshot($filename, __DIR__.'/../..');
163     }
... lines 164 - 199
```

In the scenario add:

35 lines | features/web/product_admin.feature

```
... lines 1 - 20
21    Scenario: Add a new product
... lines 22 - 25
26      And I save a screenshot to "shot.png"
27      And I fill in "Product Name" with "Veloci-chew toy"
... lines 28 - 35
```

Run it!

```
$ ./vendor/bin/behat features/product_admin.feature:20
```

The modal opens and it still fails at the "New Product" step. But *now* we have a fancy new shot.png file at the root of the project that shows exactly what things looked like when the test failed. Woah.

Saving Screenshots on Failure

If you're using continuous integration to run your tests - which you should be! - this can help you figure out *why* a test failed, which is normally pretty hard to debug. By using the hook system - something like @AfterScenario - you could automatically save a screenshot on every failure. Check out our blog post about that: Behat on CircleCI with Failure Screenshots.

Anyways, remove this line and change "Product Name" back to "Name" so that the scenario passes again:

34 lines | features/web/product_admin.feature

```
... lines 1 - 20
21    Scenario: Add a new product
... lines 22 - 25
26      And I fill in "Name" with "Veloci-chew toy"
... lines 27 - 34
```

Chapter 20: Gherkin Tables: Given I have the following:

To show off a nice Behat feature, add a line to our scenario:

35 lines | features/web/product_admin.feature

```
... lines 1 - 5
6     Scenario: List available products
... line 7
8       And there are 5 products
9       And there is 1 product
... lines 10 - 35
```

We now have one step that adds 5 products and another - almost identical that adds one more product. But, the language isn't *quite* the same, so PhpStorm highlights it as an undefined step. How can we use this language but have it re-use the definition we already have?

One trick is to add a second annotation statement to that definition. And that would make it work. But there's a better way that's new to Behat3: conditional language.

Using this/that Conditional Language

Update the annotation to There is/are :count product(s) with the 's' inside of parentheses:

199 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 16
17    class FeatureContext extends RawMinkContext implements Context, SnippetAcceptingContext
18    {
... lines 19 - 81
82        /**
83         * @Given there is/are :count product(s)
84         */
85        public function thereAreProducts($count)
... lines 86 - 197
198   }
```

Now, change the end of the scenario to look for 6 products:

35 lines | features/web/product_admin.feature

```
... lines 1 - 5
6     Scenario: List available products
... lines 7 - 11
12      Then I should see 6 products
... lines 13 - 35
```

Run just this scenario:

```
$ ./vendor/bin/behat features/product_admin.feature:6
```

The new language matches both steps and we're passing. While we're here, add a proper Background and move the login step there. Remove the duplicated line from each scenario:

35 lines | features/web/product_admin.feature

```
... lines 1 - 5
  Background:
    Given I am logged in as an admin

  Scenario: List available products
    Given there are 5 products
... lines 11 - 15
  Scenario: Products show owner
    Given I author 5 products
... lines 18 - 22
  Scenario: Add a new product
    Given I am on "/admin/products"
... lines 25 - 35
```

Using Gherkin TableNodes

Next, I want to add a test for this "Is published" flag: if a product is *not* published, it has a little x icon. If it *is* published it has a ✓. I want to make sure these are showing up correctly. Right now, all of the products are unpublished.

Woo! Time for a new scenario!

43 lines | features/web/product_admin.feature

```
... lines 1 - 21
  Scenario: Show published/unpublished
... lines 23 - 43
```

But this time, we can't just say "Given 5 products exist" because we need to control the published flag. But we can use a new trick, add:

43 lines | features/web/product_admin.feature

```
... lines 1 - 22
    Given the following products exist:
... lines 24 - 43
```

End the line with a colon and below, build a table just like we did earlier with scenario outlines. Give it two headers: "name" and "is published": I'm making these headers up: you'll see how I use them in a second. Call the first product "Foo1" and make it published. Call the second "Foo2" and make it *not* published:

43 lines | features/web/product_admin.feature

```
... lines 1 - 22
    Given the following products exist:
      | name | is published |
      | Foo1 | yes          |
      | Foo2 | no           |
... lines 27 - 43
```

Ok, keep going on the scenario:

43 lines | features/web/product_admin.feature

```
... lines 1 - 26
    When I go to "/admin/products"
    # todo
... lines 29 - 43
```

I'll stop here for now and add the missing Then line that looks for the published flag later. Try this by running only this scenario:

```
$ ./vendor/bin/behat features/product_admin.feature:21
```

Copy the new function into FeatureContext:

209 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 97
98      /**
99       * @Given the following products exist:
100      */
101     public function theFollowingProductsExist(TableNode $table)
102     {
... lines 103 - 105
106     }
... lines 107 - 209
```

The TableNode Object

Ah, but this looks different: it saw that the step had a table below it and passed us a TableNode object that represents the data in the table after it. Let's iterate over the object and dump each row out to see what happens:

209 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 100
101     public function theFollowingProductsExist(TableNode $table)
102     {
103         foreach ($table as $row) {
104             var_dump($row);
105         }
106     }
... lines 107 - 209
```

Science! Rerun the test:

```
$ ./vendor/bin/behat features/product_admin.feature:21
```

Each row is printed as an associative array using the header and the value for that row. How cool is that?

To save some time, I'll copy some code that creates Product objects:

220 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 102
103         foreach ($table as $row) {
104             $product = new Product();
105             $product->setName($row['name']);
106             $product->setPrice(rand(10, 1000));
107             $product->setDescription('lorem');
108
109             if ($row['is published'] == 'yes') {
110                 $product->setIsPublished(true);
111             }
112
113             $this->getEntityManager()->persist($product);
114         }
... lines 115 - 220
```

This adds some duplication to my FeatureContext - shame on me. In a real project, I want you to be a little more careful.

In this scenario, we won't give each product an author because we don't need that for what we're testing. Call setName() passing it $row['name']. Then if ($row['is published'] == 'yes'), add $product->setIsPublished(true);.

The is published with a space in the middle is on purpose: I want a human to be able to read the scenarios. And other than super geeks like you and I, is published reads a lot better than is_published. And, "yes" for published is more expressive than putting a 1 or 0. In FeatureContext, we translate all of that to code.

Fetch the entity manager and flush the changes at the bottom:

220 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 115
116         $this->getEntityManager()->flush();
... lines 117 - 220
```

Great!

Cool this passes! But don't get too excited: we don't have a Then statement yet.

Chapter 21: Finding inside HTML Tables

To finish the scenario, we need a Then that's able to look for a check mark on a specific row. The check mark icon itself is an element with a fa fa-check class.

There is no built in definition to find elements inside of a specific row. So let's describe this using natural language. How about:

43 lines | features/web/product_admin.feature

```
... lines 1 - 27
28      Then the "Foo1" row should have a check mark
... lines 29 - 43
```

Execute Behat to get that definition printed out for us:

```
$ ./vendor/bin/behat features/product_admin.feature:21
```

When you're feeling *really* lazy, you can add a --append-snippets flag and Behat will put the definitions inside of the FeatureContext class for you:

```
$ ./vendor/bin/behat features/product_admin.feature:21 --append-snippets
```

Change arg1 to be rowText:

231 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 118
119     /**
120      * @Then the :rowText row should have a check mark
121      */
122     public function theProductRowShouldShowAsPublished($rowText)
123     {
... lines 124 - 127
128     }
... lines 129 - 231
```

Ok, this is a bit harder. First, we need to find a row that contains that $rowText and *then* look inside of just *that* element to see if it has a fa-check class in it.

Start by finding via CSS, $row = $this->getPage()->find('css'). For the selector, use table tr: and then the contains pseudo selector that looks for some text inside. Pass %s and set the value using sprintf():

231 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 123
124         $row = $this->getPage()->find('css', sprintf('table tr:contains("%s")', $rowText));
... lines 125 - 231
```

$row will now be the first tr containing this text, or null. It's not perfect: if $rowText had some bad characters in it, the selector would fail. But this is my test so I'll be lazy until I can't. Add assertNotNull() function:

231 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 124
125         assertNotNull($row, 'Cannot find a table row with this text!');
... lines 126 - 231
```

Finally, assert that the row's HTML has a fa-check class inside of it with assertContains():

231 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 126
127             assertContains('fa-check', $row->getHtml(), 'Could not find the fa-check element in the row!');
... lines 128 - 231
```

Moment of truth:

```
$ ./vendor/bin/behat features/product_admin.feature:21
```

We're green! Now, let's get even harder.

Chapter 22: Clicking a Row in a Table (i.e. Complex Selectors)

Time for a real challenge! Deleting products, it's actually a bit harder than you might think. Here comes some curve balls -- eh eh like that baseball pun?

We need a delete button to remove individual products. BDD Time! Start with the scenario:

54 lines | features/web/product_admin.feature

```
... lines 1 - 29
30    Scenario: Deleting a product
... lines 31 - 54
```

To delete a product, we need to start with one in the database. In fact, if we start with two products, we can delete the second and check that the first is unaffected.

Add a Given that's similar to the one from the "show published/unpublished" scenario, but with a slight difference:

54 lines | features/web/product_admin.feature

```
... lines 1 - 30
31      Given the following product exists:
32        | name |
33        | Bar  |
34        | Foo1 |
... lines 35 - 54
```

Since we *only* care about the name, we don't need to bother with adding an "is published" row: keep things minimal! Create a product Bar and another Foo1. Man, those dinos can't wait to get a hold of such interesting product names!

54 lines | features/web/product_admin.feature

```
... lines 1 - 34
35      When I go to "/admin/products"
... lines 36 - 54
```

This is where it gets tricky: we'll have two rows in our table that both have 'delete' buttons, but I only want to click "Delete" on the *second* row. Add another step to do that:

54 lines | features/web/product_admin.feature

```
... lines 1 - 35
36      And I click "Delete" in the "Foo1" row
... lines 37 - 54
```

Then, it would be super to see a flash message that confirms that the product was deleted. Make sure Foo1 no longer appears in this list of products. And double check that Bar was *not* also deleted:

54 lines | features/web/product_admin.feature

```
... lines 1 - 36
37      Then I should see "The product was deleted"
38      And I should not see "Foo1"
39      But I should see "Bar"
... lines 40 - 54
```

This is the first time we've seen But. But, it has the same functionality as And: it extends a Then, When or Given and sounds natural.

Try it! Run just this scenario:

```
$ ./vendor/bin/behat features/product_admin.feature:42
```

Copy the new function into FeatureContext:

239 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 129
    /**
     * @When I click :arg1 in the :arg2 row
     */
    public function iClickInTheRow($arg1, $arg2)
    {
        throw new PendingException();
    }
... lines 137 - 239
```

Change arg1 to linkText and arg2 to rowText:

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 128
    /**
     * @When I click :linkText in the :rowText row
     */
    public function iClickInTheRow($linkText, $rowText)
    {
... lines 134 - 138
    }
... lines 140 - 254
```

This isn't the first time we've looked for a row by finding text inside of it. Let's re-use some code.

Make a new private function findRowByText() and give it a $linkText argument:

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 241
    /**
     * @param $rowText
     * @return \Behat\Mink\Element\NodeElement
     */
    private function findRowByText($rowText)
    {
... lines 248 - 251
    }
... lines 253 - 254
```

Copy the two lines that find the row and return $row. That'll make life a little bit easier:

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 247
        $row = $this->getPage()->find('css', sprintf('table tr:contains("%s")', $rowText));
        assertNotNull($row, 'Cannot find a table row with this text!');

        return $row;
... lines 252 - 254
```

Now use $this->findRowByText($rowText); in the original method and also in the new definition:

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 121
    public function theProductRowShouldShowAsPublished($rowText)
    {
        $row = $this->findRowByText($rowText);
... lines 125 - 126
    }
... lines 128 - 131
    public function iClickInTheRow($linkText, $rowText)
    {
        $row = $this->findRowByText($rowText);
... lines 135 - 138
    }
... lines 140 - 254
```

Consider the row found!

Finding Links and Buttons in a Row

To find the link, we don't want to use css: $linkText is the *name* of the text: what a user would see on the site. Instead, use $row->findLink() and pass it $linkText:

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 131
    public function iClickInTheRow($linkText, $rowText)
    {
        $row = $this->findRowByText($rowText);

        $link = $row->findLink($linkText);
... lines 137 - 138
    }
... lines 140 - 254
```

I'll repeat this one more time for fun. you can find *three* things by their text: links, buttons and fields. Use findLink(), findButton() and findField() on the page *or* individual elements to drill down to find things. Add assertNotNull($link, 'Could not find link '.$linkText); in case something goes wrong. Finally click that link!

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 136
        assertNotNull($link, 'Cannot find link in row with text '.$linkText);
        $link->click();
... lines 139 - 254
```

We haven't done any coding yet, but the scenario is done. Run it!

```
$ ./vendor/bin/behat features/product_admin.feature:42
```

It fails... but not in the way that I expected. It says

> Undefined index: is published in FeatureContext line 110.

That's happening because - this time - we don't have the 'is published' column in our table. But on line 110, we're assuming it's always there:

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 100
    public function theFollowingProductsExist(TableNode $table)
    {
... lines 103 - 108
            if ($row['is published'] == 'yes') {
                $product->setIsPublished(true);
            }
... lines 112 - 116
    }
... lines 118 - 254
```

That's fine: I like to start lazy and assume everything is there. When I need the steps to be more flexible, I'll add more code. Add an isset('is published') so if it's set *and* equals yes, we'll publish it:

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 108
            if (isset($row['is published']) && $row['is published'] == 'yes') {
                $product->setIsPublished(true);
            }
... lines 112 - 254
```

Rerun this now.

```
$ ./vendor/bin/behat features/product_admin.feature:42
```

It fails with:

> Undefined variable: rowText in FeatureContext line 256.

Hmm that sounds like a Ryan mistake. Yep: I meant to call this variable $rowText:

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 241
    /**
     * @param $rowText
     * @return \Behat\Mink\Element\NodeElement
     */
    private function findRowByText($rowText)
    {
... lines 248 - 251
    }
... lines 253 - 254
```

Now we've got the proper failure: there is no link called Delete.

Let's code for this! Remember, do as little work as possible.

Coding the Delete

Add a new deleteAction() and a route of /admin/products/delete/{id}. Name it product_delete. We could get fancy and add an @Method annotation that say that this will only match POST or DELETE requests. Let's keep it simple for now:

66 lines | src/AppBundle/Controller/ProductAdminController.php

```
... lines 1 - 50
    /**
     * @Route("/admin/products/delete/{id}", name="product_delete")
     * @Method("POST")
     */
    public function deleteAction(Product $product)
    {
... lines 57 - 63
    }
... lines 65 - 66
```

And instead of adding $id as an argument to deleteAction(), I'll be even lazier and type hint the Product so that Symfony queries for it for me.

Now, remove the $product, flush it, set a success flash message that matches what's in the scenario and finally redirect back to the product list route:

66 lines | src/AppBundle/Controller/ProductAdminController.php

```
... lines 1 - 56
        $em = $this->getDoctrine()->getManager();
        $em->remove($product);
        $em->flush();

        $this->addFlash('success', 'The product was deleted');

        return $this->redirectToRoute('product_list');
... lines 64 - 66
```

To add the delete link, find list.html.twig and add a column called Actions. Since you should POST to delete things, add a small form in each row, instead of a link tag. Make the form point to the product_delete route and add method="POST". And instead of having fields, it only needs a submit button whose text is "Delete". Add some CSS classes to make it look nice - don't get too lazy on me:

73 lines | app/Resources/views/product/list.html.twig

```
... lines 1 - 19
            <table class="table table-striped">
                <thead>
                    <tr>
... lines 23 - 26
                        <th>Actions</th>
                    </tr>
                </thead>
                <tbody>
                    {% for product in products %}
                    <tr>
... lines 33 - 38
                        <td>
                            <form action="{{ path('product_delete', {'id': product.id} ) }}" method="POST">
                                <button type="submit" class="btn btn-small btn-link">Delete</button>
                            </form>
                        </td>
                    </tr>
                    {% endfor %}
                </tbody>
            </table>
... lines 48 - 73
```

Perfect!

Try it!

```
$ ./vendor/bin/behat features/product_admin.feature:42
```

Click/Follow Links, Press Buttons

Hmmm, it fails in the *same* spot:

> And I click "Delete" in the "Foo1" row.

Either something is wrong with the way we wrote the code, there's an error on the page or we're not even on the right page. Right now, we can't tell.

Since it's failing on the "I click" line, hold command and click to see its step definition function. Var dump the $row variable to make sure we're finding the row we expected:

255 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 131
    public function iClickInTheRow($linkText, $rowText)
    {
        $row = $this->findRowByText($rowText);

        var_dump($row->getHtml());
... lines 137 - 139
    }
... lines 141 - 255
```

The other thing we can do is temporarily make this an @javascript scenario and add a break:

55 lines | features/web/product_admin.feature

```
... lines 1 - 28
29    @javascript
30    Scenario: Deleting a product
... lines 31 - 34
35      When I go to "/admin/products"
36      And break
... lines 37 - 55
```

Try it again:

```
$ ./vendor/bin/behat features/product_admin.feature:42
```

Ah-ha! We have an exception on our page and had no idea! I forgot to pass the id when generating the URL:

73 lines | app/Resources/views/product/list.html.twig

```
... lines 1 - 39
40                      <form action="{{ path('product_delete', {'id': product.id} ) }}" method="POST">
41                          <button type="submit" class="btn btn-small btn-link">Delete</button>
42                      </form>
... lines 43 - 73
```

Keep the debugging stuff in and try again:

```
$ ./vendor/bin/behat features/product_admin.feature:42
```

It stops again, but no error this time: the delete button looks fine. Press enter to keep this moving.

But it still fails! The test could not find a "Delete" link to click in the "Foo1" row. The cause is subtle: links and buttons are not the same. We click links but we press buttons. In the scenario I should say I *press* Delete instead of click:

54 lines | features/web/product_admin.feature

```
... lines 1 - 29
30    Scenario: Deleting a product
... lines 31 - 35
36      And I press "Delete" in the "Foo1" row
... lines 37 - 54
```

More importantly, inside of our FeatureContext, update to use findButton() and change the action from click to press.

254 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 128
129     /**
130      * @When I press :linkText in the :rowText row
131      */
132     public function iClickInTheRow($linkText, $rowText)
133     {
... lines 134 - 135
136         $link = $row->findButton($linkText);
137         assertNotNull($link, 'Cannot find button in row with text '.$linkText);
138         $link->press();
139     }
... lines 140 - 254
```

For clarity, change $link to $button and $linkText to $buttonText.

This should solve all 99 of our problems. I even have enough confidence to remove @javascript and the "break" line. Rerun the test!

```
$ ./vendor/bin/behat features/product_admin.feature:42
```

Finally green!

Clean up the code a bit more by changing findButton() to pressButton():

250 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 131
    public function iClickInTheRow($linkText, $rowText)
    {
        $this->findRowByText($rowText)->pressButton($linkText);
    }
... lines 136 - 250
```

Remember, this shortcut also works with clickLink() and fillField().

```
$ ./vendor/bin/behat features/product_admin.feature:42
```

And it still passes. You just won the Behat and Mink World Series! I know, *terrible* baseball joke - but the world series is on right now.

Chapter 23: Tagging Scenarios in order to Load Fixtures

Remember *way* back in the beginning when we had search feature? Try running that again:

```
$ ./vendor/bin/behat features/web/search.feature
```

Huh, this one is failing now: it says that the text "Samsung Galaxy" was not found anywhere on the page. Now that you're an expert, I hope you can spot the problem: we're not adding this product at the beginning of the scenario. This worked originally because the fixtures that come with the project have a "Samsung Galaxy" product. But now that other tests have cleared the database, we're in trouble.

We *could* put some Given statements at the top to add the products. But there's another way: load the project's fixtures automatically before the scenario. This LoadFixtures class is responsible for putting in the Kindle and Samsung products.

I think that entering the data manually with the Given statements is the most readable way to do things. But, if you do load the fixtures, here's the best way. First, I *don't* want to load fixtures before every scenario. That would make my scenarios run slower, even when I don't need that stuff.

Tagging Scenarios

Instead, I need a way to *tag* scenarios to say "this one needs fixtures". Add @fixtures at the top of this scenario outline:

19 lines | features/web/search.feature

```
Feature: Search
  In order to find products dinosaurs love
  As a website user
  I need to be able to search for products

... lines 6 - 8
  @fixtures
  Scenario Outline: Search for a product
... lines 11 - 19
```

That's called a tag, and you can put many as you want, separating each by a space. At first, adding a tag does nothing except for the magic @javascript that changes to use a JavaScript driver.

Running things Before a Tagged Scenario

But in FeatureContext you can add an @BeforeScenario method that's *only* executed when a scenario has a certain tag. Make a new public function loadFixtures(). Inside, just to see if it's working, put var_dump('GO!');. Above, put the normal @BeforeScenario:

258 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 42
    /**
     * @BeforeScenario @fixtures
     */
    public function loadFixtures()
    {
        var_dump('GO!');
    }
... lines 50 - 258
```

Here's the trick: after this, add @fixtures. Now, this will only run for scenarios tagged with @fixtures. To prove that, re-run our search.feature:

```
$ ./vendor/bin/behat features/web/search.feature
```

There's our 'GO!'. Now run the authentication.feature:

```
$ ./vendor/bin/behat features/web/authentication.feature
```

This passes with no var_dump(). Perfect!

Loading the Fixtures

One way to execute the fixture is by running the doctrine:fixtures:load command. I use a different method that gives me more control. Add $loader = new ContainerAwareLoader() and pass it the container:

263 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 47
48      public function loadFixtures()
49      {
50          $loader = new ContainerAwareLoader($this->getContainer());
... lines 51 - 53
54      }
... lines 55 - 263
```

Now, point to the exact fixtures objects that you want to load. There are two methods available: loadFromDirectory() or loadFromFile(). Move up a few directories and load from src/AppBundle/DataFixtures:

263 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 50
51          $loader->loadFromDirectory(__DIR__.'/../../src/AppBundle/DataFixtures');
... lines 52 - 263
```

That should do it!

Next, create an $executor = new ORMExecutor() and pass it the entity manager:

263 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 51
52          $executor = new ORMExecutor($this->getEntityManager());
... lines 53 - 263
```

A purger is the second argument, which you only need if you want to clear out data. We're already doing that, so I'm not going to worry about it here. Finally type $executor->execute($loader->getFixtures()) and pass true as the second argument:

263 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 52
53          $executor->execute($loader->getFixtures(), true);
... lines 54 - 263
```

This says to not delete the data, but to append it instead.

Ok, run search.feature:

```
$ ./vendor/bin/behat features/web/search.feature
```

It fails for a completely different reason. Things are never boring here! This is a unique constraint violation because it's not clearing out the data before loading the fixtures. This is a funny edge case. Because the new @BeforeScenario is near the top and the other for clearing the data is lower, they're being run in that order. Move these @BeforeScenarios up top and keep them in the order that you want:

263 lines | features/bootstrap/FeatureContext.php

```
... lines 1 - 35
    /**
     * @BeforeScenario
     */
    public function clearData()
    {
        $purger = new ORMPurger($this->getContainer()->get('doctrine')->getManager());
        $purger->purge();
    }

    /**
     * @BeforeScenario @fixtures
     */
    public function loadFixtures()
... lines 49 - 263
```

Back to the terminal and run this sucker again!

```
$ ./vendor/bin/behat features/web/search.feature
```

Pop the champagne people, it passes! It clears the data and *then* loads the fixtures. And life is super awesome!

Running Tagged Scenarios

There's another benefit to tagging scenarios. But first, if you ever need some details about the behat executable, run it with a --help flag to get all the info:

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

One of the options is tags:

> Only execute the features or scenarios with these tags

Well that's sweet. So we could say: --tags=fixtures and it will only execute scenarios tagged with fixtures:

```
$ ./vendor/bin/behat --tags=fixtures
```

Or, we can get real crazy and say that we want to run all scenarios *except* the ones tagged with @fixtures by using the handy ~ (tilde) character:

```
$ ./vendor/bin/behat --tags=~fixtures
```

behat -vvv

One more tip! If something goes wrong, there's also a verbosity option that will show you the full stack trace. Just add -v:

```
$ ./vendor/bin/behat --tags=~fixtures -v
```

Hey, that's all! Hop in there, celebrate behavior driven development, create beautiful tests and sleep better at night!

See ya next time!