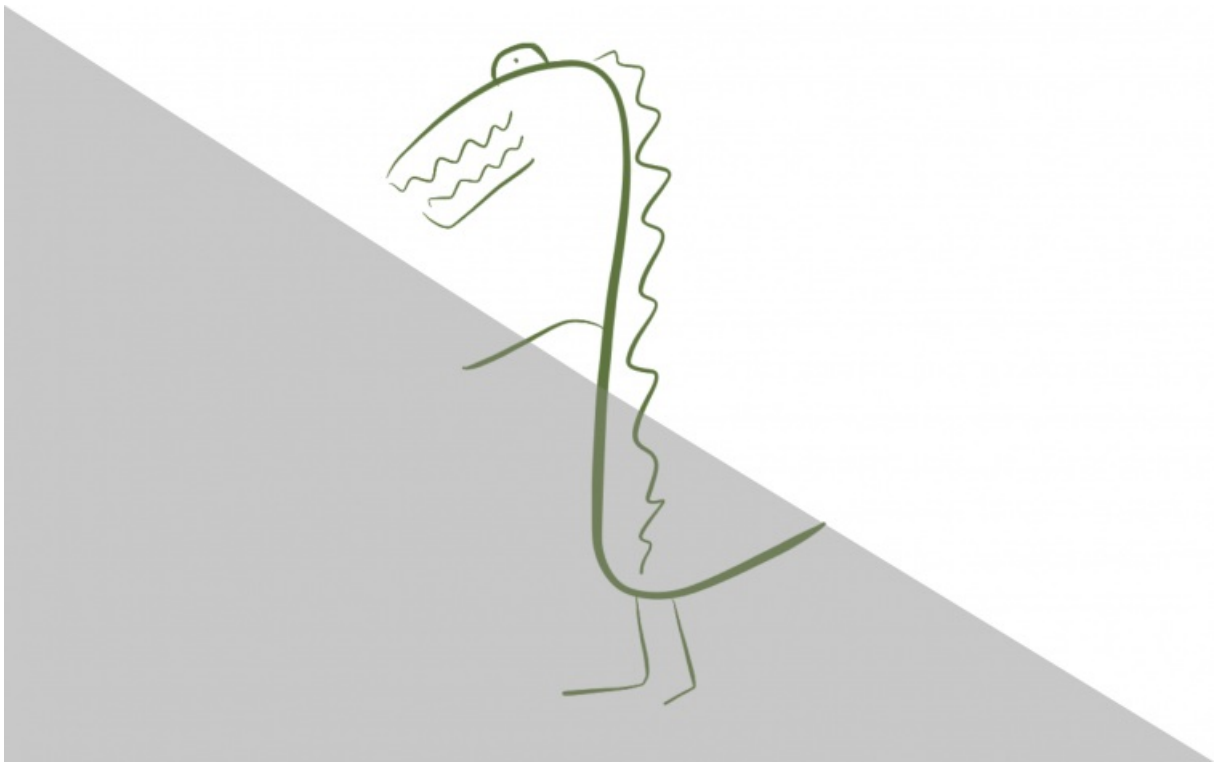
BDD, Behat (version 2.5), Mink and other Wonderful Things



With <3 from SymfonyCasts

Chapter 1: Introduction

INTRODUCTION¶

Hey! Welcome to the tutorial that we're calling "All about the World of Behat". Our goal is simple: to understand the Behavior-Driven Development philosophy and master two tools - Behat & Mink - that will make you a functional-testing legend.

Tip

Behat and Mink have been developed by the open source community and are led by our friend and yours: Konstantin Kudryashov - aka @everzet: http://twitter.com/everzet. He was a huge help in the creation of this course!

Why test your application? Well imagine you are running Jurassic Park, you need to know that adding the new Pterodactyl exhibit won't turn off the electric fence around the velociraptor pen. Don't have any tests? Good luck - they know how to open doors.

Getting good at practicing behavior-driven development - or BDD - means more than learning a new tool, it will change your entire development process for the better. Imagine a world where communication on your team is perfect, you always deliver *exactly* what your client wanted, electricity on the velociraptor fence never goes down and chocolate ice cream is free. Ok, we can't promise all of that, but we'll see how BDD makes developing fun all over again.

In this first part, I'm going to show you how to install everything you need and write your first feature and scenarios. After that, we'll back up to learn more about each important part: scenarios & features, step definitions, advanced Mink, and other really important topics.

Installation¶

Before we begin, let's install a few libraries. Start by creating a new directory and adding a `composer.json` file.

Note

If you're testing an existing application, do all of this inside your project directory.

Composer is a tool that helps download external libraries into your project. If you're not familiar with it, it's totally ok! We have a free course that explains it.

Note

Watch "The Wonderful World of Composer Tutorial" at http://bit.ly/KnpU-Composer

We'll be downloading Behat, Mink, and a few other related libraries into our project. To make this easy, we've prepared a gist with exactly what you need to add to your `composer.json` .

```
{
    "require": {
        "behat/mink": "1.4@stable",
        "behat/mink-goutte-driver": "*",
        "behat/mink-selenium2-driver": "*",
        "behat/behat": "2.4@stable",
        "behat/mink-extension": "*"
    },
    "minimum-stability": "dev",
    "config": {
        "bin-dir": "bin/"
    }
}
```

Tip

If you're using Behat in a Symfony2 project, you'll also want to include the Symfony2 Extension.

Next, download Composer by going to GetComposer.org and clicking download. Copy one of the two code blocks, depending if you have `curl` installed, and paste into the terminal. This downloads a standalone `composer.phar` executable.

Tip

Remember, we talk a lot more about composer in "The Wonderful World of Composer Tutorial" at http://bit.ly/KnpU-Composer

Next, tell Composer to download the libraries we need by running `php composer.phar install`

```
$ php composer.phar install --prefer-dist
```

We'll fast-forward through this thrilling process as it downloads each library and places it into a new *vendor/* directory. When it's finished, you'll also notice a new `bin/` directory with a `behat` file in it. This is the Behat executable, and you'll use it to run your tests and get debug information.

Next, create a `behat.yml` file at the root of the project. When Behat runs, it looks for a `behat.yml` file, which it uses for its configuration.

Tip

For more information about the `behat.yml` configuration file, see *Configuration - behat.yml* .

We'll use it to activate MinkExtension, which is like a plugin for Behat. Also, we're going to test Wikipedia, so use it as the `base_url` . If you're testing your application, use its local base URL instead. Hopefully you'll join us for the rest of this course, where we'll go into greater detail.

```
default:
  extensions:
    Behat\MinkExtension\Extension:
      goutte:    ~
      selenium2: ~
      base_url: http://en.wikipedia.org/
```

Note

If you're using Behat with Symfony2, you should also activate the *Symfony2 Extension* that you added to `composer.json` :

```
default:
  extensions:
    # ... the MinkExtension code
    Behat\Symfony2Extension\Extension: ~
```

To get the project ready to use Behat, run `php bin/behat --init` . This creates a `features/` directory and a `bootstrap/FeatureContext.php` file inside of it.

Note

If you're using Behat in Symfony2, run the command for a specific bundle. A `Features` directory will be created in that bundle, with a similar structure. If the directory is created at the root of your project, delete it and double-check that you've activated the `Symfony2Extension` in the `behat.yml` file:

```
$ php bin/behat @EventBundle --init
```

Open this file and make it extend `MinkContext` instead of `BehatContext` :

```
// ...
use Behat\MinkExtension\Context\MinkContext;

class FeatureContext extends MinkContext
{
    .. ///
}
```

Later on, we'll learn more about Behat and Mink individually, and the importance of the `MinkContext` class will make more sense.

Woo! With all that installing and configuring behind us, let's get to locking down the raptor cage!

Writing Features and running tests¶

The Behat and Mink libraries are most commonly used to test web applications. You describe a feature in a human-readable syntax called Gherkin, then execute these as tests. The best way to see this in action is to take your DeLorean back to the past a few years and imagine that Jimmy Wales has asked you to build Wikipedia.org. Yes, we know this site actually exists, but we're going to *describe* its behavior and run some functional tests against it.

First, forget about tests. Our goal is to describe the feature. We're going to describe the Wikipedia search, so create a `search.feature` file in the `features` directory. The language in this file is called Gherkin and you start by describing the feature using a specific, four-line syntax. This defines the business value of the feature, who will benefit from it, and a short description. So, when John Hammond comes to you with a big idea, your first goal should be to try to describe it using these four lines. Writing good feature descriptions is really important, and we'll spend more time on this later.

```
Feature: Search
  In order to find a word definition
  As a website user
  I need to be able to search for a word
```

Each feature has many scenarios, which describe the specific behavior of the feature. Each scenario has 3 sections. `Given` which details the starting state of the system, `When` which includes the action the user takes, and `Then` which describes what the user sees after taking action. In this scenario, we're searching for an exact article that matches.

```
Feature:
  # ...

  Scenario: Search for a word that exists
    Given I am on "/wiki/Main_Page"
    When I fill in "search" with "Velociraptor"
    And I press "searchButton"
    Then I should see "an enlarged sickle-shaped claw"
```

Great! In a normal application, we'd now start developing the feature until it it fits our description of its behavior. But since Wikipedia exists already, we can see the behavior in action!

Writing Features and Scenarios is great, because it helps clarify how something should work in human-readable language. But the real magic is that we can run the scenario as a functional test!

To do this, run `php bin/behat`. Behind the scenes, this reads the scenario and actually uses a real browser to go to Wikipedia, fill in the field, and click the button!

To see how this is possible, execute Behat, but pass a `-dl` option:

```
$ php bin/behat -dl
```

Behat's job is to read each line in the scenario and execute some function inside our `FeatureContext` class. Because we're using Mink, we inherit a lot of common sentences. You can use these to write tests without writing any PHP code. You can

also invent your own sentence and then create a new method in the `FeatureContext` class. We'll talk a lot more about this later.

Executing Tests that use JavaScript¶

Our first scenario ran in the background using a headless browser called Goutte. Goutte runs very fast, you know like a velociraptor, but it doesn't support Javascript. This was ok because our Scenario doesn't rely on any JavaScript functionality. But what if it did? Can we test things that use JavaScript?

Of course! And with Behat & Mink, it's incredibly easy. First, download Selenium Server, which is just a jar file that can live anywhere on your computer. Start Selenium at the command line by running `java -jar` followed by the filename.

```
$ java -jar selenium-server-standalone-2.28.0.jar
```

Now for the magic. To make this one scenario execute using Selenium instead of Goutte, add an `@javascript` tag above the scenario. Now just re-run your Behat tests using the same command as before:

```
$ php bin/behat
```

Magically, a browser opens up, surfs to Wikipedia, fills in the field and presses the button. This is the most powerful feature of Mink: you can run some tests using Goutte and other tests - that require JavaScript - in Selenium simply by adding the `@javascript` tag.

Digging into Gherkin, Behat and Mink¶

We now have a project using Behat & Mink, and our first feature file and scenario. Using a bunch of built-in english sentences, we're able to write tests without any work at all.

But to really get good, we need to dive deeper to find out how to write really solid Feature files, how to create your own custom sentences, how to master Mink to do really complex Browser tasks, and much more. So, keep going!

Chapter 2: Behavior-Driven Development

BEHAVIOR-DRIVEN DEVELOPMENT¶

Once upon a time, you probably didn't write any tests. We know, we've been there, it's not a happy place, but we all start somewhere. Eventually, you started writing some tests, and even read about or tried "Test-Driven Development". The idea is simple, write your tests first, and *then* write code until your tests pass. It helps clarify your goals before you spend time developing. You also know that as soon as your tests all pass, you should stop working! This makes it harder to over-perfect your code.

Behavior-driven development, or BDD, is the next evolution. It's similar to TDD except that instead of writing tests first, we'll create written descriptions of the *behavior* of a feature. Dan North - the father of all of this BDD stuff - once wrote that 'Behaviour' is a more useful word than 'test'. Of course! Writing tests is great, but before we do any work, we need to understand the exact *behavior* of the feature we're building.

There are two styles of BDD - SpecBDD and StoryBDD. Roughly speaking, Spec is used for writing unit tests. In PHP, a wonderful library called PHPSpec exists to help with this style. In this course, we're talking about the other type, StoryBDD, which is typically used for functional testing. In an ideal world, you'll use both styles.

If nothing else, BDD aims to solve the great problem of communication. Every project has a lot of players: developers, a client, project managers, velociraptors, pterodactyl and Samuel L. Jackson. As computer scientists, the development process is usually a bit of a goat rodeo, where nobody really understands the full goals or behavior of what's being built.

BDD aims to fix this by giving us a standard language for describing these features. We'll also follow a workflow that will help make the whole process from "great idea" to development much more sane:

1. **Define** the business value of the features
2. **Prioritize** features by their business value
3. **Describe** them with readable scenarios
4. And only then - **implement** them

As a developer, this might look like boring business-talk. But I've used this countless times to break a big idea into smaller pieces written in clear language. And for a developer, getting clear directions rocks!

Ok, all the theory is behind us, let's get to Gherkin, the language of BDD!

Chapter 3: Gherkin

GHERKIN¶

Writing Features (Defining Value)¶

Gherkin is the language used to describe a feature and the scenarios that define its behavior. It originally came from Cucumber, the Ruby-equivalent of Behat and is just meant to be a natural, but structured feature story.

The feature template should look familiar: it consists of four lines that define the business value and "user role":

```
Feature: {custom title}
  In order to {benefit/value of the feature}
  As a {user/role who will benefit from this feature}
  I need to {short feature description}
```

The first line starts with `Feature`, followed by a short title. This line should quickly highlight the purpose of this feature, but otherwise isn't too important.

The next two lines, however, are *very* important. First, the `In order to` line defines the *value*. Why should we build this feature? Why is it important? Will it bring us more visitors or keep those visitors safe from dinosaur attack? The next line - starting with `As a` defines *who* will benefit from this value. Is it the admin user? Our normal web user? A defenseless park guest? If you have a hard time writing these first two lines, it's possible that this feature just isn't a good idea. After all, if we're going to spend time and money building something, shouldn't it have some value for a specific person?

Finally, the last line - starting with `I need to` - is a short description of the types of actions the user will be able to take once this feature is complete.

Since an example is worth a thousand words, let's look at a few. Pretend that a big client idea has just been given to you, and it's your job to break it into smaller pieces. From the 4-step process in the last chapter, our first step is to **Define** the business value. In other words, create the four-line feature for each big part of the idea.

Suppose you hear "The site needs to be readable in French". The feature might look like this:

```
Feature: i18n
  In order to read the news in French
  As a French user
  I need to be able to switch locale
```

The value of the feature is clear: to be able to read news in French. The user that benefits from the feature is any French user. The last line details the types of things the user needs to do to get this business value. The whole feature description is simple - we'll add more detail in the next step.

Imagine also that the same site needs a news admin panel:

```
Feature: News admin panel
  In order to maintain a list of news
  As a site administrator
  I need to be able to add/edit/delete news
```

For the "value", we could say "In order to edit news". But is editing news actually the true value? Instead, let's write "In order to maintain a list of news". The user who's benefiting is our "site administrator". This makes more sense - ultimately our site administrators want to be able to maintain the list of news that shows up on the site. This is the true *business* value of the feature - the web interface we'll build is just the tool to do that.

Let's do one more example. This time, imagine that park security wants to control park fences from a mobile app, while vacationing thousands of miles away.

```
Feature: Remote fence control API
  In order to control fence security from anywhere
  As an API user
  I need to be able to POST JSON instructions that turn fences on/off
```

The person benefiting in this case is our API user. This highlights another reason why the "user" or "role" is so important: every line in a feature is written from this person's point of view and using the technical level of that person. This is *really* important, so I'll say it again. The entire feature file is written from the first person point of view of the user or role and should use language that's *only* as technical as that user understands. In this example, an API user understands the meaning of "JSON instructions". But if our role were "a park guest", we would avoid technical language like this. When we start writing scenarios, this means that you should never include CSS selectors: you understand what a CSS selector means, but your generic "web user" definitely does not.

The reason behind this is simple. The *only* reason we're spending money to build the feature is to benefit this one user type. If we can't even explain the feature using their language, then our feature is either too technical for that user, has no business value, or actually benefits some other user. It's also helpful to imagine that this user is actually requesting the feature from you, using their own language. In the real world, keeping the language simple also means that you can write features and then send them back to the client for approval.

For fun, let's look at a bad example of a feature. Suppose we've decided to put delicious humans in front of dinosaurs to entertain them while in captivity:

```
Feature: Delicious humans
  In order to be entertained
  As a dinosaur
  I need to be able to watch delicious humans pass by me all day
```

I love this example, because it sounds like something a big group of managers might come up with. The problem is that "seeing delicious humans all day" probably does not actually entertain dinosaurs. If you think that you're building this feature for their benefit, you're fooling yourself. This might very well be a good feature, but the business value is that the company will make money from park tickets, and the person benefiting from that is definitely not your dinosaur.

Prioritizing¶

Now that we've broken the big idea down into 3 features, we can prioritize which we should work on first. And since we've focused on business value, this is easy: just choose the feature that has the most. Alternatively, if you need to make your admin users happy immediately, you might choose features that benefit those users. We'll start with the news admin panel.

Prioritizing might not be something you normally do, but now it's easy. You can make sure you repair the T-Rex fence before you send your first group of visitors into the park.

Writing Scenarios¶

Once you've chosen a feature, it's time to write scenarios that describe each part of it. As we saw earlier, each scenario follows a very specific pattern. Start by giving it a name.

```
Feature: News admin panel
# ...

  Scenario: List available news
```

The body of a scenario is made up of three different parts: `Given`, `When` and `Then`. The first is `Given`, which describes the initial state of the system for the scenario. This is the *only* place where you can describe things that the user can't do. In this case, the "site administrator" can't magically put 5 news entries in the database, but that's ok. To have more than one `Given` statement, start the next line with `And`.

The second part of each scenario is `When`, which describes the actual action that this user is taking.

Finally, `Then` is used to describe what our user can see at the end of the scenario.

```
Feature: News admin panel
# ...

  Scenario: List available news
    Given there are 5 news articles
    And I am on "/admin"
    When I click "News"
    Then I should see 5 news items
```

The exact language you use in your scenarios is up to you - just make sure to follow the `Given`, `When`, `Then` format. Each line in the scenario is called a “step”, and should plainly describe what the user is doing and seeing.

```
Feature: News admin panel
# ...

  Scenario: List available news
    # ...

  Scenario: Add a new news entry
    Given I am on "/admin/news"
    When I click "New entry"
    And I fill in "Title" with "Alan Grant does not endorse the park!"
    And I press "Save"
    Then I should see "Your article has been saved"
```

Note

Technically speaking, there is no difference between `Given`, `When`, `Then` or `And` - Behat will process these steps completely the same.

If we didn’t go any further, we would at least have a standard way of describing our features. Writing scenarios also makes you think through each feature in more detail. When you’re finished, you’ve got a blueprint for exactly what you need to develop, written in language that your client can understand.

Next, we’ll use Behat to execute each Scenario as a test.

Chapter 4: Behat

BEHAT¶

Installing Behat¶

To really learn what Behat does, let's hop back in our DeLorean and pretend that we're writing the UNIX `ls` command. I'll create a new directory with a new `composer.json` file. This time, we'll install *only* Behat by following Behat's Quick Tour:

```
{
    "require": {
        "behat/behat": "2.4.*@stable"
    },
    "minimum-stability": "dev",
    "config": {
        "bin-dir": "bin/"
    }
}
```

Just like before, initialize the project by running `php bin/behat --init`. This creates the same `FeatureContext.php` class as earlier. In this case, things are simple enough that we don't need a `behat.yml` file.

Feature, Scenarios and Custom Step Definitions¶

Our project is setup, so let's create our first feature file called `ls.feature`. Remember to focus on the business value of the `ls` command:

```
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
```

Next, let's write some scenarios! Suppose that Linus Torvalds said to us:

```
If you have two files in a directory, and you're running the command - you
should see them listed.
```

Let's turn this into our first scenario. Remember that we're following the `Given`, `When`, `Then` format, but using natural language:

```
Feature: ls
# ...

  Scenario: List 2 files in a directory
    Given I have a file named "john"
    And I have a file named "hammond"
    When I run "ls"
    Then I should see "john" in the output
    And I should see "hammond" in the output
```

The goal of Behat is to let you execute your scenarios as tests. So let's try it! But this time, instead of running and passing, Behat prints out some methods and regular expressions. Copy these into your `FeatureContext` class:

```
/**
 * @Given /^I have a file named "([^"]*)"$/
 */
public function iHaveAFileNamed($argument1)
{
    throw new PendingException();
}

/**
 * @When /^I run "([^"]*)"$/
 */
public function iRun($argument1)
{
    throw new PendingException();
}

/**
 * @Then /^I should see "([^"]*)" in the output$/
 */
public function iShouldSeeInTheOutput($argument1)
{
    throw new PendingException();
}
```

Behat works by reading each step, or line, in your scenario and executing a method in `FeatureContext`, which is called a “step definition”. This is done by matching the step to the regular expressions above each method. Behat was also smart enough to generate wildcards in the regex: the quoted values are passed as arguments to the methods. This makes it easy to create re-usable steps.

Our job now is to fill in the body of each method. To check the output in the last method, we can create a new `output` property on the class and store the output there. This is a common trick when you need information between different steps. Finally, to sandbox our test, we’ll create and move into a `test/` directory:

```
private $output;

public function __construct()
{
    // this actually creates 2 test directories inside of each other!
    // the reason is subtle, and we'll fix this soon
    mkdir('test');
    chdir('test');
}

/** @Given /^I have a file named "([^"]*)"$/ */
public function iHaveAFileNamed($file)
{
    touch($file);
}

/** @When /^I run "([^"]*)"$/ */
public function iRun($command)
{
    exec($command, $this->output);
}

/** @Then /^I should see "([^"]*)" in the output$/ */
public function iShouldSeeInTheOutput($string)
{
    if (array_search($string, $this->output) === false) {
        throw new \Exception(sprintf('Did not see "%s" in the output', $string));
    }
}
```

When we run `bin/behat` again, it works! As each step is read, each method is executed.

Hooks!¶

But when we run Behat again, it blows up. If we scroll up, it makes sense. Each test creates a `test/` directory, but never cleans it up. To fix this, create a new method in `FeatureContext` that reverses the setup work:

```
public function moveOutOfTestDir()
{
    chdir('..');
    if (is_dir('test')) {
        system('rm -r '.realpath('test'));
    }
}
```

Behat creates a new `FeatureContext` object for each scenario that it runs, which means that the `__construct` method is run before every scenario. To tell Behat to run our clean method *after* each scenario just add an `AfterScenario` annotation:

```
/**
 * @AfterScenario
 */
public function moveOutOfTestDir()
{
    chdir('..');
    if (is_dir('test')) {
        system('rm -r '.realpath('test'));
    }
}
```

While we're at it, let's also move the setup code into a method that's tagged with `BeforeScenario`:

```
/**
 * @BeforeScenario
 */
public function moveIntoTestDir()
{
    mkdir('test');
    chdir('test');
}
```

Tip

If you're wondering why we didn't just use `__construct` and `__destruct`, the answer is that these methods behave slightly differently than tagging methods with `@BeforeScenario` and `@AfterScenario`.

Run Behat twice more to let the new methods clean things up. Now our test is passing perfectly every time.

Using PHPUnit assert functions¶

If you have PHPUnit installed, then you can uncomment out a few lines at the top of your test to make life easier. Once you've done this, you have access to a bunch of PHPUnit assert functions. We can use one of them, `assertContains` to make our test a bit nicer on the eyes:

```
/**
 * @Then /^I should see "([^"]*)" in the output$/
 */
public function iShouldSeeInTheOutput($string)
{
    assertContains(
        $string,
        $this->output,
        sprintf('Did not see "%s" in the output', $string)
    );
}
```

The Second Scenario¶

We've written one scenario, so let's try another! This time Linus tells us:

```
If you have one file and one directory, and you run the
command - you should see them both listed too.
```

Hopefully, writing scenarios is getting easy:

```
Feature: ls
# ...

  Scenario: List 2 files in a directory
  # ...

  Scenario: List 1 file and 1 directory
    Given I have a file named "john"
    And I have a dir named "ingen"
    When I run "ls"
    Then I should see "john" in the output
    And I should see "ingen" in the output
```

Just like before, run `bin/behat`, copy in the missing step definition, and implement it. And with almost no work, this new scenario passes!

Background¶

We now have two working scenarios, but a little bit of duplication. Specifically, each scenario starts with the same `Given I have a file named "john"`. To fix this, add a `Background` before both scenarios:

```
Feature: ls
# ...

  Background:
    Given I have a file named "john"

  Scenario: List 2 files in a directory
    And I have a file named "hammond"
    When I run "ls"
    Then I should see "john" in the output
    And I should see "hammond" in the output

  Scenario: List 1 file and 1 directory
    And I have a dir named "ingen"
    When I run "ls"
    Then I should see "john" in the output
    And I should see "ingen" in the output
```

Background is dead-simple, but really useful! When we re-run the test, each line in the background is executed before each scenario. Our scenarios are executed exactly like before, but without the duplication!

In fact, Behat has more cool tricks, including *scenario outlines*, more hooks like `BeforeScenario`, a way to organize your scenarios called tags, and much more. We'll see more of these powerful tricks a bit later.

Chapter 5: Mink!

MINK¶

Now for something totally different! You've just learned how Behat allows you to execute your Gherkin scenarios by reading each step and executing a function that has a regular expression matching it. There are more nice features to learn about, but Behat is just that simple.

Mink is a totally different library that helps you command a browser using a really nice PHP interface. It has nothing to do with Behat, but will integrate with it nicely later. Mink by itself is one of my *favorite* PHP libraries, so it deserves to get some of its own time.

To keep things simple, let's keep building on the same project we've started. To install Mink, just add it to your `composer.json` file. You should also include two more entries for the Goutte and Selenium2 drivers. You'll see why these are important in a second:

```
{
    "require": {
        "behat/behat": "2.4.*@stable",
        "behat/mink": "1.4@stable",
        "behat/mink-goutte-driver": "*",
        "behat/mink-selenium2-driver": "*",
    },
    "minimum-stability": "dev",
    "config": {
        "bin-dir": "bin/"
    }
}
```

To actually download the libraries, run `php composer.phar update behat`.

Imagine if you could control a browser using PHP code - telling the browser to go to pages, click on links, fill out form fields, and grab text or other details from each page. That's Mink. And since we've installed it via Composer, we can just create a new file, require `vendor/autoload.php`, and start playing with it:

```
<?php

require __DIR__.'/vendor/autoload.php';
```

The Driver¶

Mink is a small, but potent library that has exactly 4 important objects. The first is the driver, which will make more sense later. For now, instantiate a new `GoutteDriver`:

```
use Behat\Mink\Driver\GoutteDriver;

$driver = new GoutteDriver();
```

This says that we want our "browser" to actually use Goutte, a "headless" browser, which is a fancy word for a browser that just makes background CURL requests. This is important object #1, and it's actually not very important. So... forget I even mentioned it at all!

The Session¶

Important object #2 - and the first we care about - is the Session! You can think of the Session as your browser and use it the

same way:

```
$session = new \Behat\Mink\Session($driver);
```

It has methods like `visit`, `getCurrentUrl`, `back`, `setCookie` and a few more. In fact, let's google for the Mink API docs. I'm not always a huge fan of API docs, but Mink is a small library, and usually all you really need to know is what methods you can call on our 4 important objects.

Let's use the `Session` to go to jurassicpark.wikia.com and print out the status code and the URL:

```
$session->start();

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

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

To try this out, run the script from the command line. Success! Behind the scenes, this just made a *real* HTTP request to jurassicpark.wikia.com and then printed the status code and the current URL.

The Page (DocumentElement)¶

Important object #3 is the Page, which you can get by calling `$session->getPage()`. If the `Session` is your browser, then this object represents the HTML DOM of the current page. You could also think of it as the jQuery object, because we'll use it to traverse down our page and find different elements.

The tricky thing with the Page is that its an instance of `DocumentElement`, which doesn't sound like a Page at all! When you look at its API docs, make sure to check out all of the methods on this class as well as those it inherits from its base classes.

The easiest thing to do with the page is get its HTML or text. Let's use it to print out the beginning of the jurassicpark.wikia.com homepage:

```
$page = $session->getPage();

echo "First 160 chars: ".substr($page->getText() , 0, 160) . "\n";
```

Try the script again. Yea! We're killing it!

The NodeElement and finding via CSS¶

Getting the text of the page is great, but a more more powerful feature of the page is to find and traverse elements using CSS. To do this, use either the `find` method to get one element or `findAll` to get an array of matched elements:

```
$anchorEle = $page->find('css', 'h3 a.title');
```

The return value of `find` is a `NodeElement`, which is our 4th and last important object in Mink! If you look at its API documentation, you'll see that it extends the same base class as `DocumentElement`, which means that it has almost all the same methods and more. For example, once you've found a `NodeElement`, you can find more elements deeper inside of it:

```
$spanEle = $anchorElement.find('css', 'span.emph');
```

With all this new knowledge, let's find the sub-link beneath "On the Wiki" and print its text:

```
$element = $page->find('css', '.subnav-2 li a');

echo "The link text is: ". $element->getText() . "\n";
```

When we run the file again, it prints out “Wiki Activity”. And when we look at the site, this is exactly what we expect. Awesome!

The Named Selector¶

So far, we’ve found elements by using CSS selectors. But there is one other way to find elements, and it’s *really* important, especially when we start using Behat and Mink together. It’s called the “named” selector, and it works by trying to find elements by matching their “text”. Let’s use it to find the link whose text is “Random page”.

First, grab an object called the `SelectorsHandler`. Next, use the same `find` method as before, but in place of `css`, use `named` as the first argument. As the second argument, pass an array with two elements: the string `link`, and the string `Random page` wrapped in an `xpathLiteral` function. Yep, this looks ugly, but it’ll all make sense in a second!:

```
$selectorsHandler = $session->getSelectorsHandler();
$element = $page->find(
    'named',
    array(
        'link',
        $selectorsHandler->xpathLiteral('MinkExtension')
    )
);
```

But first, print out the URL of the `NodeElement` and test that things are actually working:

```
echo sprintf(
    "The URL is '%s'\n",
    $element->getAttribute('href')
);
```

The best way to understand the “named” selector is to open up the class that’s behind the magic:: `NamedSelector`. At the top is a large array with keys like “link”, “field”, and “button” and next to each is a big, ugly, but fascinating xpath. In our `find` method, we asked to find a `link` matching `Random page`. This then uses the long xpath next to `link` to try to find a matching element. I’m not an xpath expert, but if you look closely, you can see that it tries to find an anchor tag whose id matches `Random page`, or which contains the text `Random page`, or whose `title` attribute contains `Random page`, or which contains an `img` tag whose `alt` attribute contains `Random page` and even more after that. Basically, this will find the anchor tag that matches the text in any possible way that we might imagine. It’s important because it’s actually very “human” it allows us to find elements using the same language as our users. For example, our user would never say `Click the anchor tag that is inside an element with class subnav-2`. In reality, the user would say `Click the "Random page" link`. The named selector finds that element.

Once you understand this, you’ll love what’s next. Both the `DocumentElement` and `NodeElement` objects have a bunch of shortcuts to find things using the named selector. For example, finding the `Random page` link is as easy as saying `findLink`:

```
$element = $page->findLink('Random page');
```

Internally, this is using the “named” selector. And there are a lot more methods just like this, such as `findButton` and `findField`. As you’ll see later, the last function is especially important, because it allows you to find fields by referring to the label for that field:

```
$firstNameInput = $page->findField('First Name');
```

Click, Dragging, and doing other things with an element (NodeElement)¶

Now that you’re a master at finding elements on a page, let’s do something with them! The `NodeElement` object has methods for just about anything you could ever think to do with a field, like `click`, `press`, `check` if the element is a checkbox or `attachFile` if it’s an upload field. If you need information, you can use methods like `getTagName` or `getAttribute`. Later, when we start testing with Selenium2, you’ll even be able to do things that rely on JavaScript like `rightClick`, `mouseOver`, and `dragTo`!

Let's use this to click on the `Random page` link and print out the new URL on the next page:

```
$element->click();

echo "Page URL after click: ". $session->getCurrentUrl() . "\n";
```

Running Tests in JavaScript¶

So far, the HTTP requests that Mink is making are done in the background using CURL. But what if the site we're browsing relies on JavaScript? That just wouldn't work at all right now.

To fix this, comment out the `GoutteDriver`, and instead use the `Selenium2Driver`:

```
use Behat\Mink\Driver\GoutteDriver;
use Behat\Mink\Driver\Selenium2Driver;

//$driver = new GoutteDriver();
$driver = new Selenium2Driver;
```

Also make sure to "stop" your Mink session at the end of your script. This wasn't needed with Goutte, but with Selenium2, the `start` function opens the browser and `stop` closes it:

```
$session = new Session($driver);
$session->start();

// everything ...

$session->stop();
```

Also, remove the status code line, as Selenium doesn't support getting the page's status code.

In your terminal, start the selenium server that we downloaded earlier. And just by changing one line of code to switch drivers, when we execute the test it now actually opens up Selenium2 and performs our actions! This is the reason Mink was created: sometimes your code can be run quickly in a headless browser like Goutte and sometimes you need something that supports JavaScript like Selenium2. Mink gives you a single, simple API that lets you write the same code, and then choose very easily how you want that code written. When we use Behat and Mink together, turning Selenium2 on and off is even easier.

The Key Points to Mink¶

And that's it for Mink! Remember that Mink is really just 4 important objects: the `Driver`, `Session`, Page or `DocumentElement` and element or `NodeElement`. To find elements on a page, use the `find` or `findAll` methods with either the `css` or `named` selector. Shortcut methods like `findLink` and `findButton` make it even easier to use the named selector. And once you've found the element you need, do something with it - like calling the `click` method or getting its text via `getText`.

Chapter 6: Behat and Mink

BEHAT + MINK¶

Now that you're dangerous with Behat and an absolute expert in Mink, let's put them together!

Behat has a plugin system called "extensions", and we'll install a `MinkExtension`, which makes Behat and Mink play together like best friends! On the MinkExtension Documentation Page, copy the `composer.json` entry, paste it into your file, and then run `php composer.phar update behat/mink-extension` to download the new library.

One thing we haven't mentioned yet is Behat configuration. By default, it assumes a lot of things - like that our features live in a `features/` directory and that the `FeatureContext` class is found in `features/bootstrap/`. To configure this and a lot more, you can create a `behat.yml` file in the root of your project or in a `config/` directory. We need this file now because it's used to activate Behat extensions, like our `MinkExtension`. Copy the base configuration from the `MinkExtension` documents and also add another `selenium2` line. This activates it and says that we'll be using the `goutte` and `selenium2` drivers. For now, remove the `base_url` line.

Note

If you're using Behat in Symfony2, you should also install the Symfony2Extension and activate it here.

Accessing Mink and Built-in Definitions¶

Our goal is to get access to the Mink Session object from within our `FeatureContext` class. If we did that, we could write Behat steps like `Given I go to "/contact"` and very easily use Mink to command the browser accordingly. The `MinkExtension` helps us do this, but gives us a few options.

First, we can simply implement the `MinkAwareInterface`. This forces us to have two methods:: `setMink` and `setMinkParameters`. The `Mink` object is a container that holds a pre-built `Session`, and if we set it an a private property, we could access the Session later. When Behat starts, the `MinkExtension` looks to see if our `FeatureContext` class implements this interface. If it does, it calls `setMink` and passes us exactly what we need.

If you think that's too much work, it's your lucky day! The second option is to extend the `RawMinkContext` class, which implements this interface for you and gives you some nice shortcut methods like `getSession`.

But wait there's more! Before we look at the third option, go to the command line and ask Behat to print out the current list of all built-in definitions:

```
php bin/behat -dl
```

Not surprisingly, it prints out the few custom definitions that we wrote earlier to test the `ls` command, but nothing else. Finally, make your `FeatureContext` extend `MinkContext`. This has all the niceness of `RawMinkContext`, but with one big surprise. Print your definitions again to see that we've suddenly inherited a long list of really useful statements like `Given I am on` and `When I fill in "label" with "value"`. We just got a bunch of awesome free stuff.

So where did all this free stuff come from? The answer is in the `MinkContext` class that we just extended. If you open this up, you'll see all the regular expressions and functions that fuel these. This is a great place to look if you're writing a custom definition and want some help on exactly how to accomplish something.

Adding Web Scenarios¶

Let's copy in the first scenario that we wrote all the way back in chapter 1. For now, use the full URL to Wikipedia. We'll fix this in a few minutes. Just like before, the scenario passes without writing any custom step definitions, and now it makes sense why: every step in our scenario matches up with one of the regular expressions that we inherited.

Let's try another scenario. Suppose someone says to you:

>If you are on the main page and you fill in the search field with >"Tyrannosaurus Bill" and press "search", then you should

see “Search results”

Since Tyrannosaurus’s name is Rex and not Bill, this scenario outlines what happens when you search for an article that doesn’t exist. This time, the scenario is super-quick to write.

```
Scenario: Searching for a page that does NOT exist
  Given I am on "http://en.wikipedia.org/wiki/Main_Page"
  When I fill in "search" with "Tyrannosaurus Bill"
  And I press "searchButton"
  Then I should see "Search results"
```

And once again, without any PHP code, our scenario passes.

Background and Scenario Outline¶

Two scenarios and no code yet, we’re doing great! Let’s clean up a few things by reducing duplication. First, use the Background strategy we learned earlier to remove the identical `Given` step on each scenario.

```
Feature: Search
  # ...

  Background:
    Given I am on "http://en.wikipedia.org/wiki/Main_Page"

  Scenario: Searching for a page that does exist
    When I fill in "search" with "Velociraptor"
    And I press "searchButton"
    Then I should see "an enlarged sickle-shaped claw"

  Scenario: Searching for a page that does NOT exist
    When I fill in "search" with "Tyrannosaurus Bill"
    And I press "searchButton"
    Then I should see "Search results"
```

Besides the `Given` part, the remainder of the scenarios are also very similar. In these cases, we can leverage something called “Scenario Outlines”. This Behat shortcut basically lets you replace any part of a scenario with a variable. If we replace the search term and the expectation with variables, then we can use a table to collapse our two scenarios into a single scenario outline. As an extra bonus, some editors will even clean up your tables for you.

```
Background:
  Given I am on "http://en.wikipedia.org/wiki/Main_Page"

Scenario Outline: Searching for a specific page
  When I fill in "search" with "<search>"
  And I press "searchButton"
  Then I should see "<expectation>"

  Examples:
    | search           | expectation                  |
    | Velociraptor     | an enlarged sickle-shaped claw |
    | Tyrannosaurus Bill | Search results             |
```

When we execute the feature, it looks a little different, but passes just like before. Use “Scenario Outlines” whenever you want to test a number of similar user interactions and outcomes.

behat.yml: base_url, Parameters, and Profiles¶

The domain is not part of the behavior that we’re describing or testing and could change depending on if we’re testing locally or on a staging server. In other words, let’s get it out of our feature file!

```
Background:
  Given I am on "/wiki/Main_Page"
```

The secret to doing this is in the `behat.yml` file under a key called `base_url`. By putting the domain here, it will be automatically prefixed to our URLs.

```
default:
  extensions:
    Behat\MinkExtension\Extension:
      goutte:    ~
      selenium2: ~
      base_url: http://en.wikipedia.org/
```

But this isn't magic, in fact this trick is done quite simply. First, any value you put here is available in your `FeatureContext` class by calling the `getMinkParameter` function:

```
/** @BeforeScenario */
public function beforeScenario()
{
    var_dump($this->getMinkParameter('base_url'));
}
```

Open up the `MinkContext` class again and notice that every reference to a URL is first passed to a `locatePath` function. This function holds the magic behind how the `base_url` works, and as you can see, it's actually really simple. Whenever you refer to a URL in your `FeatureContext`, just remember to wrap the URL in this function. And if you want even more magic, you can always override this function and add whatever magic you want.

If you're curious about what other MinkExtension options are available, check out its documentation. Alternatively, if you open a class called Extension inside the library, you'll find a large configuration tree that highlights all of the possible values, including less-known capabilities like telling Selenium2 all of your desired capabilities.

behat.yml Parameters and Profiles¶

If you want to pass your own configuration into Behat, you can do that beneath a different key: `context` and `parameters`. In this spot, you can pass whatever you want.

```
default:
  context:
    parameters:
      foo: bar
      important_things: [one, two, apple]

  extensions:
    # ...
```

The values are passed into the constructor of the `FeatureContext` class, which you can store as properties and use later however you want:

```
private $fooConfig;

public function __construct(array $parameters)
{
  $this->fooConfig = $parameters['foo'];
}
```

To see what other meaningful options you can pass beneath `context`, see the `behat.yml` part of the documentation.

There's also another trick called profiles. We could use it to redefine the `base_url` value, for example.

```
default:
  extensions:
    Behat\MinkExtension\Extension:
      base_url: http://en.wikipedia.org/
      # ...

fr:
  extensions:
    Behat\MinkExtension\Extension:
      base_url: http://fr.wikipedia.org/
```

Tip

Every profile inherits the configuration from the `default` profile and then overrides it.

To execute a test with a certain profile, just use the `p` option.

```
php bin/behat -p fr
```

The French Wikipedia is a bit different so the test fails, but you get the idea.

Implementation Details: don't include CSS Selectors!¶

We have one big problem still with our scenario: we're referring to both the search box and the search button by their HTML name attribute. This is a very bad practice, because this isn't something that our user can see, it's an implementation detail that might change during development. These invisible changes aren't behavior changes, so our feature shouldn't need to change. Including CSS or other technical details is very common for beginners, but a very bad practice. Fortunately, it's easy to fix!

Replace the two offending lines with natural language that the user might actually say. Since there isn't any text on the button, a natural way to express these are `When I fill in the search box with "<search>"` and `And I press the search button`. These aren't built-in definitions, I'm just inventing what sounds natural.

```
# ...

Scenario Outline: Searching for a specific page
  When I fill in the search box with "<search>"
  And I press the search button
  Then I should see "<expectation>"

  Examples:
    # ...
```

Execute Behat so that it will give us the code snippets we need to fill in. In fact, by using the `--append-snippets` flag, Behat will append the code blocks for us. Just when you thought we couldn't get lazier, another shortcut!

```
php bin/behat features/search.feature --append-snippets
```

To fill in these methods, we have two options: let's do the hard way first. The Mink Session is available by saying `$this->getSession()` and the page via `$this->getSession()->getPage()`. Since you'll need the page all the time, let's go ahead and make a shortcut method to get it. I'm being careful with my PHPDoc so that my IDE gives me autocomplete:

```
/**
 * @return \Behat\Mink\Element\DocumentElement
 */
protected function getPage()
{
    return $this->getSession()->getPage();
}
```

Back to work! We can use CSS to find the search box by using the `find` method. Alternatively, we can use the `named` selector by taking advantage of the `findField` shortcut method. Once we have the field, just call `setValue` on it:

```
/**
 * @When /^I fill in the search box with "([^"]*)"$/
 */
public function iFillInTheSearchBoxWith($searchTerm)
{
    $ele = $this->getPage()->findField('search');
    $ele->setValue($searchTerm);
}
```

For the button, we'll use the `findButton` method and then call `press` on it:

```
/**
 * @Given /^I press the search button$/
 */
public function iPressTheSearchButton()
{
    $this->getPage()->findButton('searchButton')->press();
}
```

Notice that with both of these, it's ok to include CSS selectors and other technical details inside the `FeatureContext` class. We only want to hide them from the Feature file, which should describe the features behavior at the technical level of our user.

It works! One great thing about this is that we can use these new definitions on all of our future scenarios: we only have to do this work once.

Metasteps¶

As easy as that last step was for such a Mink expert, there's an even easier solution: metasteps. Metasteps let us use Gherkin language right inside a custom step definition. Get rid of all of that Mink code, create a new instance of a `When` object, and re-use the same Gherkin step language we had earlier. There are also `Given` and `Then` classes, but all three do the exact same thing, so use whichever you want:

```
// ...
use Behat\Behat\Context\Step\Given;
use Behat\Behat\Context\Step\When;
use Behat\Behat\Context\Step\Then;

// ...

public function iFillInTheSearchBoxWith($searchTerm)
{
    return new When(sprintf(
        'I fill in "search" with "%s"',
        $searchTerm
    ));
}

public function iPressTheSearchButton()
{
    return new When('I press "searchButton"')
}
```

Metasteps are a really simple way to take the technical stuff out of your Feature without needing to write any real code.

But they're also very useful in another situation. Imagine for a second that we need to login, so we write something like `Given I am logged in`.

```
Background:
  Given I am on "http://en.wikipedia.org/wiki/Main_Page"
  And I am logged in
```

In reality, logging requires several steps, but we don't want to repeat these on each scenario: we don't care how you login, we only care that it happens.

In this case, metasteps become *very* useful because you can actually return a whole array of steps that should be executed:

```
/**
 * @Given /^I am logged in$/
 */
public function iAmLoggedIn()
{
    return array(
        new Given('I am on "login"'),
        new Given('I fill in "Username" with "Ryan"'),
        new Given('I fill in "Password" with "foobar"'),
        new Given('I press "Login"'),
    );
}
```

By writing one simple step in a scenario, you can trigger a whole group of actions to be taken. Use this often for your most commonly needed tasks. And just like with the Gherkin scenarios, there is no difference between using `Given`, `When` or `Then`: use whichever one sounds most natural to you.

Chapter 7: JavaScript

JAVASCRIPT¶

We saw earlier how Mink can be used to execute tests in a headless browser like Goutte or using a driver like Selenium2 that supports JavaScript. So how can we do this inside Behat?

Let's start by adding another scenario. Suppose someone says:

>If you fill in the search field with "Tyran" and wait for the suggestion >box to appear, then you will see "Tyrannosaurus" in it.

This tests the search autocomplete on Wikipedia, which obviously requires JavaScript. But don't worry about that yet, first worry about writing the scenario. Remember to use as many of our built-in definitions as possible.

```
Scenario: Searching for a page with autocompletion
  When I fill in the search box with "Tyran"
  And I wait for the suggestions box to appear
  Then I should see "Tyrannosaurus"
```

Since we're about to use Selenium2, make sure you've started the Selenium2 server, which is the JAR file we downloaded *earlier*.

```
cd /path/to/downloads
java -jar selenium-server-standalone-2.XX.X.jar
```

Tip

Replace XX.X with the real numbers in the file you downloaded.

To run the scenario in JavaScript, just add a `javascript` tag above the scenario.

```
@javascript
Scenario: Searching for a page with autocompletion
  # ...
```

When we run the test, the browser opens up, but our test doesn't pass yet: we're missing one very important step definition.

Waiting for things to happen¶

The autocomplete drop-down doesn't show up instantaneously, which is why we added the wait step to our scenario. But how can we wait for things with Mink? The answer is, well, the `wait` function!

```
/**
 * @Given /^I wait for the suggestion box to appear$/
 */
public function iWaitForTheSuggestionBoxToAppear()
{
    $this->getSession()->wait(5000);
}
```

If we run the test now, it passes, but it waits a whole five seconds to be sure the auto-complete opens. This is far from ideal - if you add a few of these waits, your tests are going to slow *way* down.

Instead, we can add a second argument to `wait`, which is some JavaScript that's run every 100 milliseconds. As soon as the

JavaScript expression returns true, Mink will stop waiting. And if it takes longer than 5 seconds, it will finally give up with an error:

```
/**
 * @Given /^I wait for the suggestion box to appear$/
 */
public function iWaitForTheSuggestionBoxToAppear()
{
    $this->getSession()->wait(
        5000,
        "$('.suggestions-results').children().length > 0"
    );
}
```

Since this JavaScript is run on your page, you can use whatever JavaScript libraries you have available. Wikipedia uses jQuery, so we can take advantage of it.

Chapter 8: You bet your Sweet App

YOU BET YOUR SWEET APP¶

You now know all the important things about Behat and Mink and how they work together. In this chapter, we'll talk about some specific challenges that you'll likely face when using Behat in your application, like how to handle data fixtures.

I've created a miniature application using Silex for us to play with. Don't worry if you're not using Silex - we'll keep the instructions here as generic, but useful as possible. You'll still have a bit of homework to do for your specific application, but we'll guide you.

Note

The Silex application we're using here is available in this screencast's code download. (FTW!)

Bootstrapping your Application in FeatureContext¶

To test our application, we'll make real HTTP requests as an outsider: surfing to pages, filling out forms, and checking how everything looks afterwards. To do this, we don't need access to our project's code, nor do we even need to be on the same server!

But when you are on the same server, it can be *very* useful to bootstrap your application inside `FeatureContext`. This allows you access to all the functions and tools that you use to do things like connect to the database, clear cache, or anything else. As we'll talk about in a second, having access to your normal tools might make clearing and preparing data pretty easy.

Exactly *how* you do this will be different for every application and framework, but it will follow the same pattern. Usually it involves requiring some core bootstrap file and possibly calling some sort of initialize function.

Tip

If you're using Symfony2, install the Symfony2Extension to bootstrap Symfony for you and give you access to the Kernel. The container is then available as `$kernel->getContainer()`.

Since we only need to bootstrap our application once per test suite, we can take advantage of the `beforeSuite` hook. Your setup - and even the hook to use - may be different:

```
private static $app;

/**
 * @BeforeSuite
 */
static public function bootstrapSilex()
{
    if (!self::$app) {
        self::$app = require __DIR__.'/../../app/bootstrap.php';
    }

    return self::$app;
}
```

For Silex, the `$app` object is the key to any functionality we may need. In your app, you may need a different variable, or you might just need to statically call a function that magically gives you access to all of your classes and functions. The best way to find out how your code should look is to google to see if others have bootstrapped your framework for testing, or to look at the code in your front controller.

In the next sections, you'll see how having access to the built-in functions

Preparing Data¶

One of the most complex issues with testing is dealing with and controlling the data you test with. If you're testing against an external server, you may not be able to control your data at all. In this case, you'll have to be a bit more careful and clever about how you write your tests. Fortunately, since we're on the same server as our application, we'll have full control over our data.

Let's start with a scenario I already prepared, which tests the list page of a product admin section.

```
Feature: Product admin
  In order to manage the content on my site
  As an admin
  I need to be able to add, edit and delete products

  Scenario: Seeing a list of existing products
    Given I am logged in as an admin
    And there are 5 products
    And I am on "/admin"
    When I follow "Products"
    Then I should see 5 rows in the table
```

In order for this scenario to work, we need to guarantee that there is an admin user in the database and we need the ability to add 5 products. If you've bootstrapped your application in `FeatureContext`, this should be possible. Execute `bin/behat` so that the missing definitions are added to our `FeatureContext`.

I'll fill in the code that my system needs to create my admin user. Notice that we didn't say `Given there is an admin user called "admin"` in our scenario. We don't really care about that detail, so we skip it and just make sure the user exists inside this definition. Next, use metasteps to actually login with this user. The exact wording will vary for your app:

```
/**
 * @Given /^I am logged in as an admin$/
 */
public function iAmLoggedInAsAnAdmin()
{
    self::$app['user_repository']->createAdminUser(
        'admin',
        'adminpass'
    );

    return array(
        new Given('I am on "/login"'),
        new Given('I fill in "Username" with "admin"'),
        new Given('I fill in "Password" with "adminpass"'),
        new Given('I press "Login"'),
    );
}
```

I'll also add the code to insert 5 products. Notice that we're not testing the actual creation of products in this scenario. We may do that later in another scenario, but for now we want to insert them as quickly as possible to test that the user sees them:

```
/**
 * @Given /^there are (\d+) products$/
 */
public function thereAreProducts($num)
{
    for ($i = 0; $i < $num; $i++) {
        self::$app['product_repository']->createProduct(
            'Sickle-shaped Claw'.$i,
            9.99+$i
        );
    }
}
```

Finally, write some custom Mink code for the final step. This step is purposefully generic, so that we can re-use it on other pages. Let's use the `find` method to find a single table, then use it again to return an array of all of the `tr` elements. Use the PHPUnit assert functions to make sure we have the right number of rows:

```
/**
 * @Then /^I should see (\d+) rows in the table$/
 */
public function iShouldSeeRowsInTheTable($rows)
{
    $table = $this->getPage()->find('css', '.main-content table');
    assertNotNull($table, 'Cannot find a table!');

    assertCount(intval($rows), $table->findAll('css', 'tbody tr'));
}
```

Great! Execute Behat. It passes! There's a lot going on behind the scenes, but the actual scenario is described with simple language.

Use Data Fixtures?¶

Another approach to loading data is using some sort of fixture, which inserts a whole set of default data. This may sound easy, but it's not a great approach for two reasons. First, loading the extra data makes your tests run a bit slower. Second - and more importantly - loading a set of data can make your scenarios less readable. In our scenario, we're being very specific about what data we have. If we loaded fixtures beforehand that contained 5 products, we might remove the `And there are 5 products` line. But now our scenario is a bit confusing - why are we expecting 5 products? Where did these products come from?

For those reasons, do your best to avoid loading fixtures. The one exception might be if you have lookup tables that contain data that never changes, and is important to your application. An example might be a table called `product_status`, with entries like `Published`, `Draft`, `Archived`. Since this data is static, it just needs to be there, so loading it before your tests is probably a good idea.

Cleaning out Data¶

Run the test again. This time, it fails spectacularly. When we try to insert a second user with the same username, a unique constraint in the database fails. As important as it is to add the data you need in a scenario, you also need to clean out data. At the beginning of each scenario, you should be able to assume that there is no data in the database. This prevents us from needing to say things like `Given there are no users` before saying `Given I am logged in as an admin`. The fact that we need to empty the user table before inserting a user is an implementation detail - not part of the feature's description.

Exactly how you handle this depends on your application, but it almost always involves another `BeforeScenario` hook. Create a new function called `clearData` and tag it with `BeforeScenario`. In here, your goal is to empty the data in the database. In reality, you can do this, or just empty the tables that you know should be cleared before each test. For now, let's clear the `user` and `product` tables:

```
/**
 * @BeforeScenario
 */
public function clearData()
{
    self::$app['user_repository']->emptyTable();
    self::$app['product_repository']->emptyTable();
}
```

We're using `BeforeScenario` here because each scenario should be independent of every other scenario. In other words, data built in one scenario shouldn't be used in the next. By clearing out the data before each one, we're helping to guarantee that independence.

Re-run the test to see that things pass once again. Regardless of how you clear it, make sure to always think about what data you have and what you're adding so that you're testing against the exact data you want.

Using the Current User and other Objects¶

Sometimes you'll write a scenario where you refer to something that was just created in the background. For example, what if we want the five new articles to be authored by us, the admin user? Lets change our scenario to reflect this:

```
Scenario: Seeing a list of existing products
  Given I am logged in as an admin
  And I author 5 products
  # ...
```

Run behat to generate this new step definition. We already know how to create products, but how can we set our user as the author? The trick is to set the current user on a private property when we login:

```
private $currentUser;

// ...

/**
 * @Given /^I am logged in as an admin$/
 */
public function iAmLoggedInAsAnAdmin()
{
    $this->currentUser = self::$app['user_repository']->createAdminUser(
        'admin',
        'adminpass'
    );

    return array(
        // ...
    );
}

/**
 * @Given /^I author (\d+) products$/
 */
public function iAuthorProducts($num)
{
    for ($i = 0; $i < $num; $i++) {
        $product = self::$app['product_repository']->createProduct(
            'Sickle-shaped Claw'.$i,
            9.99+$i
        );

        $product->author = $this->currentUser;
        self::$app['product_repository']->update($product);
    }
}
```

Once we've done this, we can use it in this definition or any other in the future. This is one of the pro tips to using Behat, and we saw it once before during the `ls` scenarios. If you need access to something between steps, just store it on a property. Scenarios should be completely independent of each other, but the steps in a scenario can be totally *dependent*.

Customizing behat.yml on each Machine¶

Finally, let's cover one more obstacle to using Behat in your project. The `behat.yml` file holds the `base_url` configuration, and based on our virtualhost configuration, this value may be different on your machine versus my machine. But how can we commit this to our repository without everyone needing to modify this file and try *not* to commit those changes?

There are a few ways to handle this, but one of them is with the `imports` configuration. In `behat.yml`, import a separate file called `behat.local.yml` and then move the `base_url` into it.

```
# behat.yml
default:
  extensions:
    Behat\MinkExtension\Extension:
      goutte:    ~
      selenium2: ~

imports:
  - behat.local.yml
```

```
# behat.local.yml
default:
  extensions:
    Behat\MinkExtension\Extension:
      base_url:  http://store.l
```

The point of this is that we'll commit `behat.yml`, but add `behat.local.yml` to our `.gitignore` file. When someone sets up the project for the first time, they'll just create this file and customize it however they need.

To make this easier, copy this file to `behat.local.yml.dist`.

```
cp behat.local.yml behat.local.yml.dist
```

This new file has no functional purpose, but you can use it to create the local file when you setup the project.

Chapter 9: The fun stuff Chapter

THE FUN STUFF CHAPTER¶

You're now a Behat and Mink pro! In this last chapter, I want to mention a few tricks about debugging, and a number of other important tips and features.

Debugging Failures¶

So what happens when a test fails and you don't know why? Let's make our product list page throw a big error and re-run the test:

```
throw new \Exception('Ah, ah, ah, you didn\'t say the magic word!');
```

It fails of course, but it's not really obvious why.

Whenever you have a failure that you need to debug, use a special built-in step called "print last response". If you forget the wording for this step, just re-run `bin/behat -dl` - you'll find it near the bottom. Place this step just above the one that fails and re-run your test:

```
Scenario: Seeing a list of existing products
  Given I am logged in as an admin
  And there are 5 products
  And I am on "/admin"
  When I follow "Products"
  And print last response
  # the "Then" below is failing right now
  Then I should see 5 rows in the table
```

It may be a bit long, but you'll now see the HTML response from the last page, just before the error. A lot of times, you'll see an error. You may even realize that you're not on the right page - the URL at the top of the output helps you figure that out. You can also copy the URL and put it in your browser, which may help you see the error yourself.

Subcontexts¶

To run this test in Selenium, just add the `@javascript` tag above the scenario. In Selenium we can still use `print last response` to debug, but there's a much better way.

First, let's talk about an idea called Subcontexts. Normally, Behat reads the annotations from our `FeatureContext` and any classes we extend, like `MinkContext`. But what if we found a really cool open source context that we wanted to use? We can only extend one other class, so how could we get the definitions from this new one?

The answer is via a subcontext. We won't show it here, but you can actually separate your definitions into as many context classes as you want. By calling `useContext` in the constructor of your main `FeatureContext`, Behat will use all the definitions and hooks from that class as well:

```
public function __construct()
{
    $this->useContext('other_context', new OtherContext());
}
```

And in fact, there *are* some open source context classes out there that you can use. One really interesting one is in the Behatch Contexts library. We're not going to install this extension, but let's browse its source code to see a few interesting context classes it has. One is the `BrowserContext`, which has some extra browser-related steps. Another is the `TableContext`, which has nice steps for dealing with tables. My only word of warning is that many of these steps require you to put CSS in

your scenario. Remember, this is a *bad* practice. I really like this extension, but because of these CSS definitions, I usually use it as a reference, rather than actually installing it.

Debugging in Selenium¶

The most interesting one is the `DebugContext`, which contains a very cool step called `I put a breakpoint`. Copy this into our `FeatureContext` file and change the text to simply read `break`. Update your scenario to use this new step instead of the `print last response`:

```
Scenario: Seeing a list of existing products
  Given I am logged in as an admin
  And there are 5 products
  And I am on "/admin"
  When I follow "Products"
  And break
  # the "Then" below is failing right now
  Then I should see 5 rows in the table
```

When we run the test, it opens up Selenium as expected. But when it hits the new step, it pauses. At the command line, you can see that it's actually waiting for us to hit "enter" before it continues. This is the best way to see what is in your actual browser and even play around with things to figure out why something is failing. When you're ready to keep going, just hit enter and the test will finish. I *love* this debugging technique.

Waiting for AJAX¶

When you're testing in JavaScript, Selenium2 is pretty good about waiting for your page to load before trying to continue and click on any elements. But when you start to load things with AJAX, you might start having problems.

We talked about this earlier when testing on Wikipedia, but let's see it again. On our test app, if you press "New Product", an AJAX call is made in the background, which causes a slight delay before the window opens. To see how this is a problem, let's write a scenario that clicks this link and creates a new product:

```
@javascript
Scenario: Add a new product via the dialog
  Given I am logged in as an admin
  And I am on "/products"
  When I follow "New Product"
  And I fill in "Name" with "New Article"
  And I fill in "Price" with "5.99"
  And I press "Save"
  Then I should see "Product created"
```

The scenario is simple, but when we run it, it fails! The "New Product" link is clicked, but since Selenium doesn't see the "Title" field immediately, it fails.

When you have these types of issues, you'll need to add a wait step. In this case, we need to wait for the dialog box to appear, so let's just say that in our scenario:

```
@javascript
Scenario: Add a new product via the dialog
  # ...
  When I follow "Create Product"
  And I wait for the dialog to appear
  And I fill in "Title" with "New Product"
  And I fill in "Body" with "Lorem Ipsum"
  # ...
```

Execute Behat so that it prints out the new definition code. Remember that waiting is done with the `wait` function, but that we only want to wait until the needed action happens. In our case we can find the Twitter Dialog element and test to see if it is visible:

```
/**
 * Wait for the twitter bootstrap dialog to appear
 *
 * @Given /^I wait for the dialog to appear$/
 */
public function iWaitForTheDialogToAppear()
{
    $this->getSession()->wait(
        5000,
        "$('.modal').is(':visible');"
    );
}
```

This will wait for 5 seconds or until the modal becomes visible. Try the test again. Egads It works! Using waits is critical to testing with JavaScript, but it's also really important that you wait for specific things to happen, not static lengths of time. If you're using consistent loading screens and dialogs, then you should be able to write and re-use just a few wait steps.

The TableNode syntax: inserting a bunch of things at once¶

Let's change our first scenario to be just a little more interesting. Right now, we're inserting 5 products and checking for 5 products. That's a great scenario, but we might also choose to insert some specific products, and then check for them directly. I'm also going to change the behavior of the application to only show published products on the list page:

```
Scenario: Seeing a list of existing products
  Given I am logged in as an admin
  And there are the following products:
    | title                        | is published |
    | The T-Rex has escaped        | yes          |
    | They can open doors...       | yes          |
    | When Dinosaurs ruled the Earth | no         |
  And I am on "/admin"
  When I follow "Products"
  Then I should see 2 rows in the table
  And I should see "The T-Rex has escaped"
  And I should not see "When Dinosaurs ruled the Earth"
```

You can already see how this scenario is now much more useful: we're not only testing that the list page works, we're testing that it shows the products it should and that it doesn't show un-published products.

The table syntax is the key here, any time you end a step with a colon, you'll create a table of data that you want to pass into your function. When we execute Behat, we'll see that the function it generates is passed a special `TableNode` object, which has all the data from that table.

With this object, we're dangerous! Using the `getHash` function, we can iterate over each row to create the products we need. One important thing to notice is that we try to keep the language in the table as natural as possible, using "is published" and "yes" or "no" instead of "true" or "false":

```
/**
 * @Given /^there are the following products:$/
 */
public function thereAreTheFollowingProducts(TableNode $table)
{
    foreach ($table->getHash() as $productData) {
        $product = self::$app['product_repository']->createProduct(
            $productData['title'],
            15.99
        );

        if ($productData['is published']) {
            $product->isPublished = true;
        }

        self::$app['product_repository']->update($product);
    }
}
```

Run the test. It still works! We're really getting good at this!

You'll also see this table syntax in one important built-in definition: `I fill in the following`. This can be used when you need to fill in a lot of fields at once, and we can use it in our product creation scenario:

```
@javascript
Scenario: Add a new product via the dialog
  # ...
  When I follow "Create Product"
  And I wait for the dialog to appear
  And I fill in the following:
    | Title | New Product |
    | Body  | Lorem Ipsum |
  # ...
```

This is a great way to fill in big forms, while keeping our scenario clean. There's also a similar syntax whenever you need to reference multi-line text. We won't talk about it here, but it's pretty easy to use.

Command-line Options: Running just one Scenario¶

The last thing I want to talk about is the many options you have when executing behat. For example, what if we only want to execute one scenario? Executing a single feature can be done easily by referencing only the filename that you want to run:

```
./bin/behat features/product.feature
```

Tip

If you're using Symfony2 and the Symfony2Extension, use the syntax `./bin/behat @SomeBundle/product.feature`.

To execute only a single scenario, just find the line number where the scenario starts, and add that to the end of the command:

```
./bin/behat features/product.feature:6
```

This is really awesome when debugging a failing scenario.

The `behat` executable has a lot of other useful options as well, which you can see by adding `--help` after the command. If you're using `behat` on a continuous integration server, you may pass a `--format=junit` option so that it outputs the JUnit XML format:

```
./bin/behat --format=junit --out=build/
```

Another useful option is `--tags`. We've seen how you can tag a scenario with `@javascript` to execute that test with Selenium. You can also invent whatever tags you want, as a way to organize your tests. Once you've done this, you can execute all the scenarios for a specific tag, all the scenarios except those with a tag, or any other logical combination you can think of:

```
./bin/behat --tags=list
```

Chapter 10: Get Testing!

GET TESTING!¶

I hope you're really excited to start using Behat and Mink in your project, we *love* to use it in ours.

Note on testing APIs¶

Behat is typically used to test websites, but it can also be used very well to test against your API's. If you want more details, see the WebApiContext class, which holds a lot of step definitions for testing your API.

Continuous Integration¶

One common question is: "Can I use Behat and Mink on my continuous integration server?". The answer is absolutely yes! In the last chapter, we talked about the `junit` format, which you can use to hook into your CI system. Beside this, there are a number of things you'll need to think about.

Tip

We now have even more information on continuous integration, specifically with Travis CI: How to use Behat and Selenium on Travis CI

First, you'll need a real database to be setup and configured for your application.

Second, your application needs to be web accessible on your CI server. Remember that Mink operates by making real HTTP requests, so your application needs to be configured with your web server so that you can at least make local requests to real URLs.

Finally, if you're using Selenium you'll need to have the Selenium server running and a utility called `xvfb` running. On a server, you don't really have a GUI, which means that you can't really open a browser. With `xvfb`, which stands for X virtual frame buffer, you can run a browser as if you were on any computer. The exact setup will vary, but remember to start `xvfb` and export your display:

```
sh -e /etc/init.d/xvfb start
export DISPLAY=:99.0
```

Until Next Time¶

Ok, start testing! And we'll see you next time!