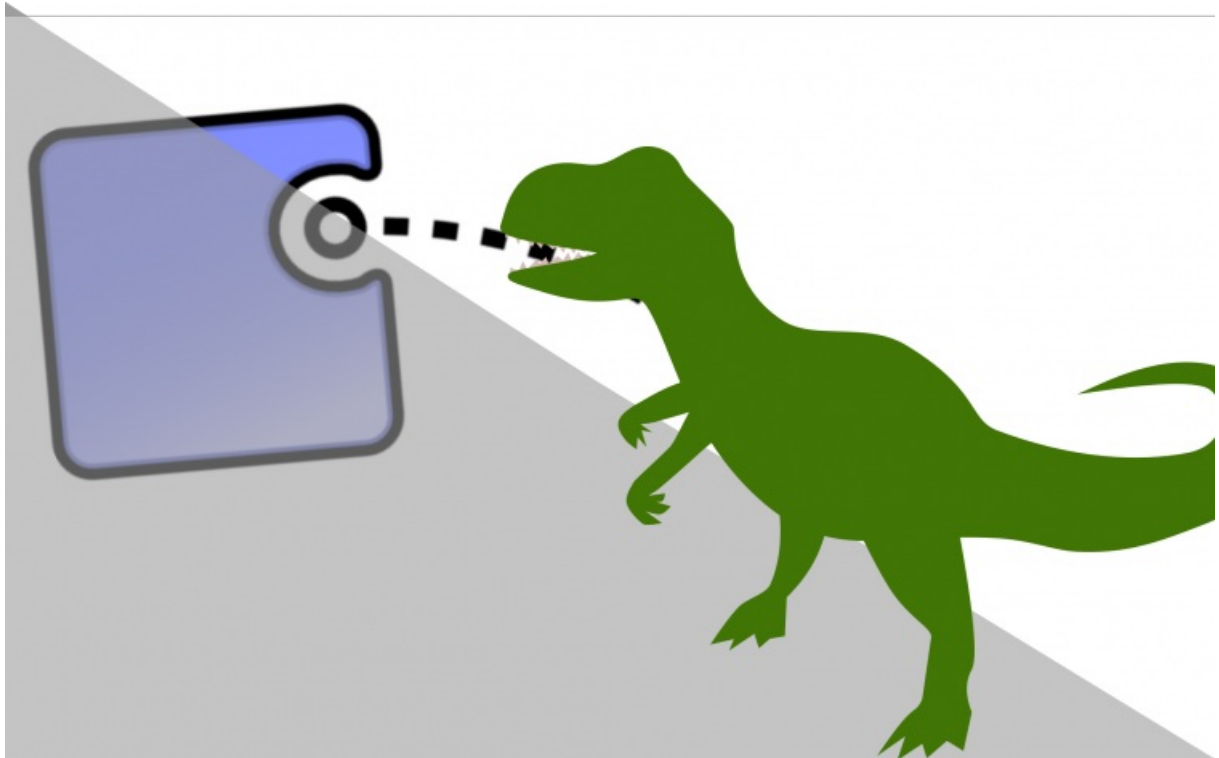


PHPUnit: Testing with a Bite



With <3 from SymfonyCasts

Chapter 1: PHPUnit: Secure the Park

Let me paint you a scary picture:

It's Friday night. It's stormy. The office is empty and you're deploying fresh code straight to production. Suddenly, an alarm! What? The fences are down!? The dinosaurs are escaping!? Somehow, your beautiful code contained a bug. And as the raptors surround you, one thought keeps coming back: if only you had written tests.


I hate bugs, I hate fixing emergencies on production, and I *especially* hate being eaten by raptors.

It's time to go pro with coding... and that means, learning to test! And not *just* because we hate running from dinosaurs. We want to write code that is thoughtfully-designed and have the ability to add *new* features with confidence.

[Hello Dinosaur Park](#)

To make this dream come alive, you should *totally* code along with me. Download the course code from this page. After unzipping the file and turning the fences back on, you should find a `start/` directory that has the same code you see here. Open up the `README.md` file for setup instructions and directions on how to catch the last boat off the island before the storm.

The last step is to find a terminal, move into your project, and run:




```
$ ./bin/console server:run
```

to start the built-in PHP web server. In your browser, go to `http://localhost:8000` to find... well... nothing! Just, "Welcome to Dinosaur Park". Instead of creating a park full of dinosaurs and *then* worrying about security... ahem... we don't have *any* code yet. We're going to build this dino factory *and* write tests all at the same time.


[Installing PHPUnit](#)

The de facto standard tool for testing in PHP is PHPUnit. Open a new terminal tab. Install it with:



```
$ composer require --dev phpunit/phpunit
```

This will obviously download the PHPUnit library into your `vendor/` directory. But mostly, you will interact with PHPUnit as an executable. When this finishes, you can now run:



```
$ ./vendor/bin/phpunit
```

Hi PHPUnit! And hello Sebastian Bergmann and other contributors! There are *no* tests yet but I already feel like we're making friends.

[Write some Tests](#)

Let's write our first test. But, uh, don't worry about *what* we're testing yet - let's just experiment a little.

Create a `tests/` directory and inside, another `AppBundle\Entity` directory. We'll talk about this structure soon, but first we have dinosaurs to contain!

Add a new PHP class: `DinosaurTest`, and give it a namespace: `Tests\AppBundle\Entity`. Make sure you *extend* a class: `TestCase` from PHPUnit.

```

14 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 2
3  namespace Tests\AppBundle\Entity;
... line 4
5  use PHPUnit\Framework\TestCase;
... line 6
7  class DinosaurTest extends TestCase
8  {
... lines 9 - 12
13 }

```

To actually make a test, create a public function called `testThatYourComputerWorks`. We're giving it that name because, inside, we're going to say `$this->assertTrue(false)`.

```

14 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 6
7  class DinosaurTest extends TestCase
8  {
9      public function testThatYourComputerWorks()
10     {
11         $this->assertTrue(false);
12     }
13 }

```

If this test *passes*, you'll know to throw your computer out of the window and buy a new one. Let's find out. To run the tests, find your terminal, and re-run PHPUnit:

```

$ ./vendor/bin/phpunit

```

Yes! It fails! My computer gets to live! It failed asserting that false is true on DinosaurTest line 11.

[Test Rules & Best Practices](#)

Ok, let's talk about the basic *rules* of writing a PHPUnit test. First, you can *technically* put your test classes *anywhere*... but.. you've gotta admit that `tests/` is a *pretty* good place. Actually, in a Symfony project, you automatically start with a `phpunit.xml.dist` file. Well, in Symfony 4, this is added when you install phpunit. We'll talk more about this file later... but PHPUnit reads this automatically. And... check it out! *It* says that our tests all live in... `tests/`. That's how PHPUnit is able to find our `DinosaurTest`.

Second, our test has a namespace... but that's not really important. In a Symfony project, your `composer.json` file has an `autoload-dev` section that basically says that anything in `tests/` should start with the namespace `Tests`. No big deal.

Let's get to the *really* important stuff, because PHPUnit *does* have a few crucial rules. First, your test class *must* extend `TestCase` *and* end in the word `Test`. Second, all of your test methods must be public and *start* with the word `test`. When you run PHPUnit, it basically looks for all classes ending in `Test` and all public functions inside *starting* with `test`.

Got it? Good... because the storm is coming, and we've got work to do. Delete the fake test. Let's start coding.

Chapter 2: Tests, Assertions & Coding

So... we don't really have much code to test yet! That might feel weird: when I started testing, I wanted to write the code first, and *then* test that it worked.

But actually, you can do it in the *opposite* order! There's a hipster methodology called Test Driven Development - or TDD - that says you should write the *tests* first and *then* your code. We'll talk more about TDD in a few minutes.

Imagining Your Future Code

But yea! That's what we're going to do. We need to use our imagination. Let's imagine that in the Entity directory, we're going to need a Dinosaur class. And each Dinosaur will have a length property, along with `getLength()` and `setLength()` methods. So, in `DinosaurTest`, we might want to test that those methods work correctly.

Actually, testing getter and setter methods is not something I *usually* do in my code... because they're *so* simple. But, it's a *great* place to get started.

Test Directory Structure

Oh, and *now* it might be a bit more obvious *why* we called this class `DinosaurTest` and put it in an `AppBundle/Entity` directory. As a best-practice, we usually make our tests/ directory match the structure of `src/`, with one test class per source class. But not all classes will need a test... and there will be some exceptions when we talk about integration and functional tests!

Testing `getLength` & `setLength`

Ok, let's test! Create a new public function `testSettingLength()`. The method needs to start with `test`, but after that, give it a clever description that will help you recognize what the method is supposed to do.

```
20 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 6
7  class DinosaurTest extends TestCase
8  {
9      public function testSettingLength()
10     {
... line 11
12
... lines 13 - 17
18     }
19 }
```

Now, even though we *don't* have a `Dinosaur` class, we're going to *pretend* like we do. Ok... `$dinosaur = new Dinosaur()`. Then, `$this->assertSame()` that zero is `$dinosaur->getLength()`.

```

20 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 6
7  class DinosaurTest extends TestCase
8  {
9      public function testSettingLength()
10     {
11         $dinosaur = new Dinosaur();
12
13         $this->assertSame(0, $dinosaur->getLength());
... lines 14 - 17
18     }
19 }

```

We <3 Assertions

This tests that - if we don't set a length - it defaults to 0. PHPUnit has a *ton* of these assert functions. Google for "PHPUnit Assertions" to find an appendix that talks *all* about them. I'd say there is a plethora of them and you'll learn them as you go, so no need to memorize all of these. There will *not* be a pop quiz at the end.

For `assertSame()`, the first argument is the *expected* value and the second is the *actual* value. `assertSame()` is *almost* the... same as the *most* popular assert function: `assertEquals()`. The only difference is that `assertSame()` also makes sure the *types* match.

And honestly, you could just use `$this->assertTrue()` for *everything*: `0 === $dinosaur->getLength()`.

But, using *specific* assert methods will give you better error messages when things fail.

Coding & Testing

Set the length: `$dinosaur->setLength(9)`. And then assert that it equals 9.

```

20 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 6
7  class DinosaurTest extends TestCase
8  {
9      public function testSettingLength()
10     {
... line 11
12
... lines 13 - 14
15         $dinosaur->setLength(9);
... line 16
17         $this->assertSame(9, $dinosaur->getLength());
18     }
19 }

```

Perfect! Our test is done! I know... kinda strange, right? By writing the test first, it forces us to think about *how* we want our Dinosaur class to look and act... instead of just diving in and hacking it together.

We *know* the test will fail, but let's try it anyways! Run:

```
$ ./vendor/bin/phpunit
```

Yay!

Adding the Dinosaur Entity

Time to make that test pass! If you downloaded the course code, then you should have a `tutorial/` directory with a `Dinosaur`

class inside. Copy that and paste it into the *real* Entity directory.

```
28 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 2
3 namespace AppBundle\Entity;
... line 4
5 use Doctrine\ORM\Mapping as ORM;
... line 6
7 /**
8  * @ORM\Entity
9  * @ORM\Table(name="dinosaurs")
10 */
11 class Dinosaur
12 {
13     /**
14      * @ORM\Column(type="integer")
15      */
16     private $length = 0;
17     ... lines 17 - 26
27 }
```

This is just a simple class with a length property. It *does* have Doctrine annotations, but that's not important! Sure, we will *eventually* be able to save dinosaurs to the database, but our test doesn't care about that: it just checks to make sure setting and getting the length works.

Let's add those methods: public function `getLength()` that returns an int. And public function `setLength()` with an int argument. Set the length property.

```
28 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
13     ... lines 13 - 17
18     public function getLength(): int
19     {
20         return $this->length;
21     }
22
23     public function setLength(int $length)
24     {
25         $this->length = $length;
26     }
27 }
```

Back in `DinosaurTest`, add the use statement. Ah, PhpStorm is as happy as a raptor in a kitchen!

```
21 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 2
3 namespace Tests\AppBundle\Entity;
... line 4
5 use AppBundle\Entity\Dinosaur;
... lines 6 - 7
8 class DinosaurTest extends TestCase
9     ... lines 9 - 20
```

Ok, find your terminal and... test!



```
$ ./vendor/bin/phpunit
```

Yes! Celebration time! This is our very *first* - of *many* passing tests!

Testing for Bugs

Let's add one more quickly: imagine that a bug has been reported! Gasp! People are saying that if they create a Dinosaur of length 15, by the time it is born and grows up, it's smaller than 15! The dinos are shrinking! Probably a good thing.

Let's add a test for this: public function testDinosaurHasNotShrunk. Start the same as before: `$dinosaur = new Dinosaur();` and `$dinosaur->setLength(15);`.

```
30 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 7
8  class DinosaurTest extends TestCase
9  {
... lines 10 - 20
21  public function testDinosaurHasNotShrunk()
22  {
23      $dinosaur = new Dinosaur();
... line 24
25      $dinosaur->setLength(15);
... lines 26 - 27
28  }
29  }
```

And *just* to make things more interesting, imagine that it's OK if the dinosaur shrinks a little bit... it just can't shrink *too* much. The guests want a thrill! In other words, `$this->assertGreaterThan(12, $dinosaur->getLength());`.

```
30 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 7
8  class DinosaurTest extends TestCase
9  {
... lines 10 - 20
21  public function testDinosaurHasNotShrunk()
22  {
... lines 23 - 26
27      $this->assertGreaterThan(12, $dinosaur->getLength(), 'Did you put it in the washing machine?');
28  }
29  }
```

You can also add an optional message as the last argument to *any* assert function. This will display when the test fails... which can sometimes make debugging easier.

Ok, try the test!

```
$ ./vendor/bin/phpunit
```

Because our code is *actually* perfect, it passes! But if you make it fail temporarily and run the test again... there's our message, along with the normal failed assertion text.

Hey! In just a few minutes, we wrote our first test and *even* used test-driven development! It's time to learn more about that... and all the different *types* of tests you can write.

Chapter 3: TDD & Unit, Integration & Functional Tests

Unit, Functional & Integration Tests

Before we get back to coding... we need to talk about just a *little* bit of theory! No, don't run away! Give me just 2-ish minutes!

So, there are actually *three* different *types* of tests, and we're going to try them all. The first is a unit test - that's what we've created. In a unit test, you test one specific function on a class. It's the *purest* way to test: you call a function with some input, and test the return value. We'll learn more about this later, but each unit test is done in *isolation*. If, for example, a class needs a database connection, we're actually going to *fake* that database connection so that we can focus on testing *just* the logic of the class itself.

The second type of test is an *integration* test... or at least, that's my name for it. An integration test is basically a unit test: you call functions and check their return values. But now, instead of faking the database connection, you'll use the *real* database connection. We'll talk about when and why this is useful later.

The third type of test is a *functional* test. In our world, this basically means that you're writing a test where you programmatically command a browser. Yep, you literally write PHP code that tells a browser to go to a page, click a link, fill out a form, click submit, and then assert that some text appears on the next page.

More on *all* of these things later.

How much to Test?

Another question is how *much* you should test. Does every function need a unit test? Does every page and every validation error of every form need a functional test? Absolutely not! That sounds worse than a raptor claw across a chalkboard!

Especially if you're new to testing, don't worry: a *few* tests is way better than none. And honestly, I think many people create *too* many tests. I follow a simple rule: if it scares me, I test it. Too many tests take extra time, add little value, and slow you down later when they fail after you've made a minor change.

In our app, we've tested the `getLength()` and `setLength()` methods. These do *not* scare me. In the real world, I would not test them.

Test-Driven Development

A few minutes ago, I mentioned the term "Test-Driven Development" or TDD. TDD breaks coding into three steps. First, create the test. Second, write the *minimum* amount of code to get that test to pass. And third, now that your tests are passing, you can safely refactor your code to make it fancier.

So, test, code, refactor: these are the steps we're going to follow. Do I *always* use TDD in my real projects? Um... yes!? No, not really: I'm not a purist. Heck, sometimes you don't even need a test! Gasp! But yea, usually, if I plan to write a test for a new feature I'm working on I'll follow TDD.

Oh, and TDD is about *more* than just making sure you have a lot of tests. As you'll see, it forces you to *think* about how you want to design your code.

Enough theory! Let's try it already!

Chapter 4: TDD in Practice

Let's put TDD into practice!

I want to add a new `getSpecification()` method to the `Dinosaur` class that will return a string description - like:

Velociraptor carnivorous dinosaur is 5 meters long

TDD says: write that test first! Add public function `testReturnsFullSpecificationOfDinosaur()`. Create a new dinosaur, but don't set any data on it. Let's test the default string: Unknown non-carnivorous dinosaur is 0 meters long should equal `$dinosaur->getSpecification()`.

```
40 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 7
8  class DinosaurTest extends TestCase
9  {
... lines 10 - 29
30  public function testReturnsFullSpecificationOfDinosaur()
31  {
32      $dinosaur = new Dinosaur();
33
34      $this->assertSame(
35          'The Unknown non-carnivorous dinosaur is 0 meters long',
36          $dinosaur->getSpecification()
37      );
38  }
39  }
```

Test done! Step 2 is to write the *minimum* amount of code to get this test to pass. In `Dinosaur`, add public function `getSpecification()`. So... what's the smallest amount of code we can write? We can just return a hardcoded string!

```
33 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 27
28  public function getSpecification(): string
29  {
30      return 'The Unknown non-carnivorous dinosaur is 0 meters long';
31  }
32  }
```

Genius! Ok, try the test!

```
$ ./vendor/bin/phpunit
```

Ha! It passes! The third step to TDD is refactor... which I don't think is needed in this case.

Wait, what? You *don't* like my hardcoded string? Looks like you're missing the last boat back to the mainland. Just kidding ... I know, returning a hardcoded string is *silly*... and I don't do this in real life. But it shows off an important thing with TDD: keep your code simple. Don't make it unnecessarily fancy or cover unnecessary use-cases. If you *do* have an edge-case that you want to cover, add the test first, and *then* code for it.

[Adding another Case](#)

Actually, let's do that now: add a new test method: `testReturnsFullSpecificationForTyrannosaurus`. I want each Dinosaur to have a *genus* - like Tyrannosaurus - and a flag that says whether or not it likes to eat people... I mean whether or not it's carnivorous. These will be used in `getSpecification()`. Create a new `Dinosaur()` and pass Tyrannosaurus, and true for carnivorous... because T-Rex's *love* to eat people.

Set its length to 12. This time, the specification should be:

Tyrannosaurus carnivorous dinosaur is 12 meters long

```
52 lines | tests/AppBundle/Entity/DinosaurTest.php
... lines 1 - 7
8  class DinosaurTest extends TestCase
9  {
... lines 10 - 39
40  public function testReturnsFullSpecificationForTyrannosaurus()
41  {
42      $dinosaur = new Dinosaur('Tyrannosaurus', true);
43
44      $dinosaur->setLength(12);
45
46      $this->assertSame(
47          'The Tyrannosaurus carnivorous dinosaur is 12 meters long',
48          $dinosaur->getSpecification()
49      );
50  }
51  }
```

[Finishing getSpecification\(\)](#)

Test done! Let's write some code! Start in Dinosaur: I'll add a `__construct()` method with a `$genus = 'Unknown'` argument and `$isCarnivorous = false`. Add these two properties to the class. I'll go to the Code menu and then to Generate - or press Command+N on a Mac - select "ORM Annotations" to add annotations above each method. We don't technically need those right now... but it'll save time later.

```

56 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 17
18 /**
19  * @var string
20  * @ORM\Column(type="string")
21  */
22 private $genus;
23
24 /**
25  * @var bool
26  * @ORM\Column(type="boolean")
27  */
28 private $isCarnivorous;
29
30 public function __construct(string $genus = 'Unknown', bool $isCarnivorous = false)
31 {
... lines 32 - 33
34 }
... lines 35 - 54
55 }

```

Down in the constructor, set both properties to their values. The default values for each argument match our first test.

```

56 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 29
30 public function __construct(string $genus = 'Unknown', bool $isCarnivorous = false)
31 {
32     $this->genus = $genus;
33     $this->isCarnivorous = $isCarnivorous;
34 }
... lines 35 - 54
55 }

```

In `getSpecification()`, we can't really fake things anymore. Return `sprintf()` and the original string, but replace the variable parts with `%s`, `%s` and `%d`.

Then pass `$this->genus`, `$this->isCarnivorous` to print carnivorous or non-carnivorous, and then `$this->length`.

```

56 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 45
46 public function getSpecification(): string
47 {
48     return sprintf(
49         'The %s %s dinosaur is %d meters long',
50         $this->genus,
51         $this->isCarnivorous ? 'carnivorous' : 'non-carnivorous',
52         $this->length
53     );
54 }
55 }

```

Perfect! Find your terminal and run the tests!

```
$ ./vendor/bin/phpunit
```

Passing! Now to step 3... refactor! And this time... I will! Let's include the word carnivorous in the string. Then below, just print non- if needed. I don't even need to *think* about whether or not I made any mistakes. Just run the tests!

```

56 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 45
46 public function getSpecification(): string
47 {
48     return sprintf(
49         'The %s %scarnivorous dinosaur is %d meters long',
... line 50
51         $this->isCarnivorous ? " : 'non-',
... lines 52 - 53
54     }
55 }

```

```
$ ./vendor/bin/phpunit
```

I love it! Testing gives you *confidence*!

Next! Let's create a DinosaurFactory - because that sounds *awesome*.

Chapter 5: Factory Testing

Ok: our dinosaur park is going to be *huge*! Like, dinosaurs *everywhere*. So we can't keep creating dinosaurs by hand. Nope, we need a DinosaurFactory!

You guys know the drill: start with the test. I think the *class* should eventually live in a Factory directory, so create a Factory folder inside tests. Then, add a new PHP class. But, wait! PhpStorm isn't auto-filling the namespace. That's annoying!

Let's fix it! Go into the PhpStorm Preferences and search for Composer. Ah, find the Composer section. This is a really cool feature - I don't know why it's not enabled by default. Click to select your composer.json path, and then make sure the "Synchronize IDE Settings" check box is checked.

Woh! The entire tests/ directory just turned green... like a dinosaur! PhpStorm reads the autoload-dev section of composer.json and *now* knows what namespace to use. Creepy.... Create a new PHP class again: DinosaurFactoryTest.

Make it extend the usual TestCase from PHPUnit. And add a new method: public function testItGrowsALargeVelociraptor().

```
21 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 7
8  class DinosaurFactoryTest extends TestCase
9  {
10     public function testItGrowsALargeVelociraptor()
11     {
... lines 12 - 18
19     }
20 }
```

Planning the Design

Ok, let's think about the *design* of this new class. There are a few dinosaurs that people just *love* - like velociraptors, tyrannosaurus and triceratops. To grow those quickly, I think it would be cool to add methods on DinosaurFactory like growVelociraptor(). Each method would already know the correct genus name and isCarnivorous value... so the *only* argument we need is \$length.

Add the Test

Awesome! Let's start using this imaginary class. First, create it: \$factory = new DinosaurFactory(). Then, \$dinosaur = \$factory->growVelociraptor() and pass in the length.

```
21 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 7
8  class DinosaurFactoryTest extends TestCase
9  {
10     public function testItGrowsALargeVelociraptor()
11     {
12         $factory = new DinosaurFactory();
13         $dinosaur = $factory->growVelociraptor(5);
... lines 14 - 18
19     }
20 }
```

Next, add some basic checks: \$this->assertInstanceOf() to make sure that this returns a Dinosaur::class instance. We can also use \$this->assertInternalType() to make sure that a string is what we get back from \$dinosaur->getGenus().

Let's also check that value exactly - it should match Velociraptor. And make sure that 5 is the length.

21 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php

... lines 1 - 7

```
8 class DinosaurFactoryTest extends TestCase
9 {
10     public function testItGrowsALargeVelociraptor()
11     {
12         ... lines 12 - 14
13     }
14 }
15 $this->assertInstanceOf(Dinosaur::class, $dinosaur);
16 $this->assertInternalType('string', $dinosaur->getGenus());
17 $this->assertSame('Velociraptor', $dinosaur->getGenus());
18 $this->assertSame(5, $dinosaur->getLength());
19 }
20 }
```

Perfect! There's one cool thing you may not have noticed: we're now *indirectly* testing some of the methods on Dinosaur. Yep, if we have a bug in `getGenus()` or `getLength()`, we'll discover that here... even if we don't have a test *specifically* for them. This is a good thing to keep in mind when trying to decide what to test. Sure, having a specific test for `getLength()` is the *strongest* way to ensure it keeps working. But since that method is pretty simple... *and* since it's indirectly tested here, that's more than enough in a real project.

Ok, step 1 of TDD is done! Let's run the test to make sure it fails:

```
$ ./vendor/bin/phpunit
```

[Coding up DinosaurFactory](#)

Yes! Let's code! Create the new Factory directory, then the class inside: `DinosaurFactory`. With TDD, writing the code is easy: we already know the method's name, its arguments and *exactly* how it should behave. Add public function `growVelociraptor`. We know this needs a `$length` argument and that it will return a `Dinosaur` object.

18 lines | src/AppBundle/Factory/DinosaurFactory.php

... lines 1 - 6

```
7 class DinosaurFactory
8 {
9     public function growVelociraptor(int $length): Dinosaur
10     {
11         ... lines 11 - 15
12     }
13 }
14 }
```

Create the new `Dinosaur` object inside and pass it `Velociraptor` and `true` for the `isCarnivorous` argument. Set the length to `$length` and return that fresh, terrifying dinosaur!

```

18 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
9      public function growVelociraptor(int $length): Dinosaur
10     {
11         $dinosaur = new Dinosaur('Velociraptor', true);
12
13         $dinosaur->setLength($length);
14
15         return $dinosaur;
16     }
17 }

```

Back in DinosaurFactoryTest, add the use statement for the new class.

```

22 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 2
3  namespace Tests\AppBundle\Factory;
... lines 4 - 5
6  use AppBundle\Factory\DinosaurFactory;
... lines 7 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
... lines 11 - 20
21 }

```

Ok, run the tests!

```

$ ./vendor/bin/phpunit

```

Woh! It fails!

Undefined method Dinosaur::getGenus()

That makes sense! And PhpStorm was trying to warn us. This is actually really cool: TDD helps you to think about what methods you need and what methods you do *not* need. We *could* have automatically added getter and setter methods for every property in Dinosaur. But, that's totally unnecessary! Less methods means less opportunity for bugs: only add methods when you need them.

Add the getGenus() method and return that property. Try the tests again:

```

61 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 55
56     public function getGenus(): string
57     {
58         return $this->genus;
59     }
60 }

```

```

$ ./vendor/bin/phpunit

```

They pass!

Refactor!

And that means it's time to refactor! Since we're going to eventually create a *lot* of dinosaurs, let's create a new private function called `createDinosaur()` with three arguments: `$genus`, `$isCarnivorous` and `$length`.

```
21 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14     private function createDinosaur(string $genus, bool $isCarnivorous, int $length)
15     {
... lines 16 - 18
19     }
20 }
```

Use those on the new `Dinosaur()`.

```
21 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14     private function createDinosaur(string $genus, bool $isCarnivorous, int $length)
15     {
16         $dinosaur = new Dinosaur($genus, $isCarnivorous);
17
18         $dinosaur->setLength($length);
19     }
20 }
```

Above in `growVelociraptor()`. return `$this->createDinosaur()` and pass `Velociraptor`, `true` and the length.

```
21 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
9      public function growVelociraptor(int $length): Dinosaur
10     {
11         return $this->createDinosaur('Velociraptor', true, $length);
12     }
... lines 13 - 19
20 }
```

And because this has a test, *it* will tell us if we made any mistakes. But I doubt that... try the test!

```
● ● ●
$ ./vendor/bin/phpunit
```

Explosion!

Return value of `DinosaurFactory::growVelociraptor` must be a dinosaur: null returned

Whooops. I forgot my return value. Let's even add a return-type for that method.

23 lines | [src/AppBundle/Factory/DinosaurFactory.php](#)

... lines 1 - 6

7 class DinosaurFactory

8 {

... lines 9 - 13

14 private function createDinosaur(string \$genus, bool \$isCarnivorous, int \$length): Dinosaur

15 {

... lines 16 - 19

20 return \$dinosaur;

21 }

22 }

This is the power of test-driven development... and testing in general.



\$./vendor/bin/phpunit

Now the tests pass.

Next, let's talk about a few *hooks* in PHPUnit that we can use to organize our test setup.

Chapter 6: Hooks: setUp, tearDown & Skipping Tests

We can *also* use the TDD refactor step to improve our tests! Eventually, we're going to have a *lot* of test methods inside `DinosaurFactoryTest`. And *each* one will need to create the `DinosaurFactory` object. If that class eventually has some constructor arguments, that's going to be a pain!

The setUp Hook

Add a new `$factory` property and give it some PHPDoc: this will be a `DinosaurFactory` object. Then - here's the magic part - create a new public function `setUp()`. Inside, set the property to a new `DinosaurFactory`.

```
36 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
11     /**
12      * @var DinosaurFactory
13      */
14     private $factory;
    ... line 15
16     public function setUp()
17     {
18         $this->factory = new DinosaurFactory();
19     }
    ... lines 20 - 34
35 }
```

Back in our test method, use the new property. Yep, this *will* work... but only thanks to a bit of PHPUnit magic. If you have a method that's exactly called `setUp`, PHPUnit will automatically call it *before* each test.

```
36 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
    ... lines 11 - 20
21     public function testItGrowsALargeVelociraptor()
22     {
23         $dinosaur = $this->factory->growVelociraptor(5);
    ... lines 24 - 28
29     }
    ... lines 30 - 34
35 }
```

If you have *multiple* test functions, that means that `setUp` will be called before each test *method*. This will make sure that the `$factory` property is a new, *fresh* `DinosaurFactory` object for *every* test. And that's *really* important: each test should be *completely* independent of each other. You never want one test to rely on something a different test set up first. Why? Because later, we'll learn how to execute just *one* test at a time - which is *really* useful for debugging.

Other Hooks: tearDown, setUpBeforeClass, etc

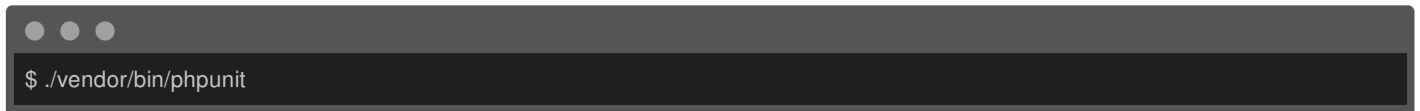
There are a few other *magic* methods like this. The most common is `tearDown()`, which is the opposite of `setUp`. It's still called once per test, but *after* the test is executed. It's meant for cleanup, and we'll talk more about it later.

Two other useful hook methods are `setUpBeforeClass()` and `tearDownAfterClass`. Instead of being called before or after

every test, these are called *once* for the *entire* class. They're less common, but if you need to setup something global or *static*, this is the place to do it.

Oh, and one last, lesser-known hook method is `onNotSuccessfulTest`. Sometimes I'll use that to print extra debugging info.

Ok, make sure the tests still pass!



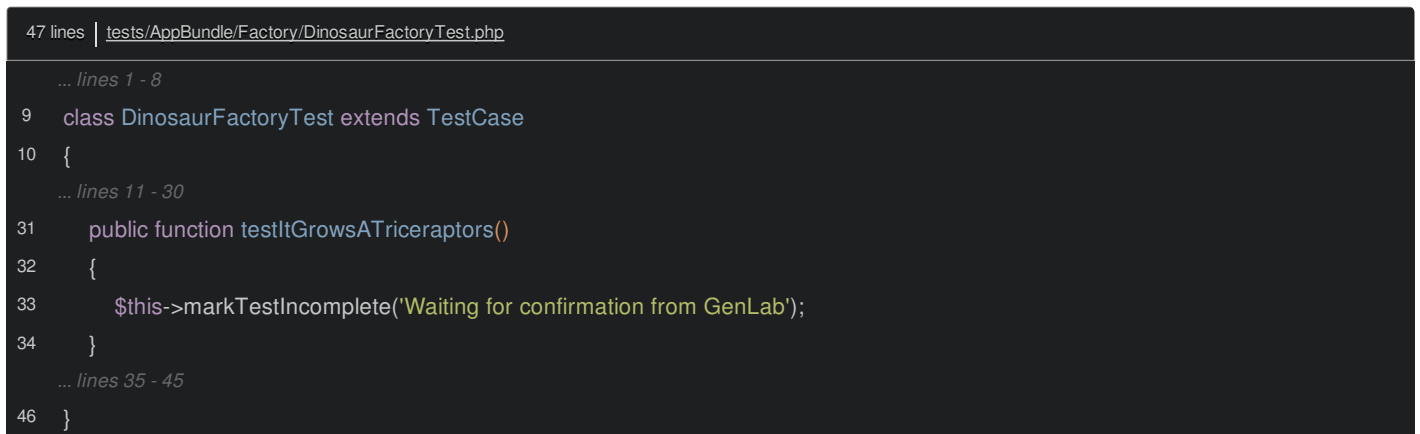
```
$ ./vendor/bin/phpunit
```

Perfect!

Marking Tests as Incomplete

Our dinosaur park guests are *really* excited about seeing some triceratops! But... we can't grow them yet - the scientists are still working on that dino DNA.

But *eventually*, we're going to add a `growTriceratops` method to `DinosaurFactory`. To make sure we don't forget about this, let's *start* the test: `testItGrowsATriceratops`. But I don't *really* want this test to exist... and fail - that's lame. Instead, add `$this->markTestIncomplete('Waiting for confirmation from GenLab')`.



```
47 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
    ... lines 11 - 30
31  public function testItGrowsATriceratops()
32  {
33      $this->markTestIncomplete('Waiting for confirmation from GenLab');
34  }
    ... lines 35 - 45
46 }
```

Try it!



```
$ ./vendor/bin/phpunit
```

Nice! It's not a failure... just a clear marker to remind us that we have work to do!

Skipping Tests

A *similar* thing you can do is *skip* tests. Try this: add a new method: `testItGrowsABabyVelociraptor()`. Create a *tiny* velociraptor - adorable! - and make sure it's length is correct.



```
47 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
    ... lines 11 - 35
36  public function testItGrowsABabyVelociraptor()
37  {
    ... lines 38 - 41
42      $dinosaur = $this->factory->growVelociraptor(1);
43
44      $this->assertSame(1, $dinosaur->getLength());
45  }
46 }
```

This will *totally* work. But let's pretend that, inside the `growVelociraptor()` method, we use some class or PHP extension that the user may or may not have installed. Check to see if some imaginary Nanny class exists. If it doesn't, we can't run the test! So mark it as skipped: there's nobody to watch the baby raptor!

```
47 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9   class DinosaurFactoryTest extends TestCase
10  {
... lines 11 - 35
36      public function testItGrowsABabyVelociraptor()
37      {
38          if (!class_exists('Nanny')) {
39              $this->markTestSkipped('There is nobody to watch the baby!');
40          }
... lines 41 - 44
45      }
46  }
```

```
$ ./vendor/bin/phpunit
```

When you run the tests now... cool! An I for incomplete and S for skipped.

I don't use `markTestSkipped()` in my own apps - it's a bit more useful when you're building some reusable library and need to write tests for optional features that use optional libraries. It's used all the time inside Symfony's core.

Next! I want to talk about my *favorite* feature in PHPUnit: data providers!

Chapter 7: Data Providers!

Park management has just *dreamt* up a new, fancy feature: they want to be able to create a new dinosaur... just by *describing* it. They want to be able to say:

```
Large friendly carnivorous dinosaur
```

and then, poof! Our DinosaurFactory will figure out *exactly* what type of dino to grow and make it.

New Feature, New Test

This means we need a new method in DinosaurFactory and *that* means we need a test. How about: `itGrowsADinosaurFromASpecification()`, where "specification" is the word we'll use for this "dinosaur description".

```
56 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9   class DinosaurFactoryTest extends TestCase
10  {
... lines 11 - 46
47      public function testItGrowsADinosaurFromSpecification()
48      {
... lines 49 - 53
54      }
55  }
```

The API for this future method will be simple: `$dinosaur = $this->growFromSpecification()` and pass it some string, like `large carnivorous dinosaur`. Maybe we should have included the word 'friendly'....meh, probably not necessary.

Now, add some assertions! Like, `$this->assertGreaterThanOrEqual()` that the dinosaur is 20 meters or longer. Actually, instead of hardcoding that value, open up the Dinosaur class and add a `const LARGE = 20`. Use that inside the test.

```
68 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11  class Dinosaur
12  {
13      const LARGE = 20;
... lines 14 - 66
67  }
```

Then, `assertTrue` that `$dinosaur->isCarnivorous()` and give that a custom failure message.

```
56 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9   class DinosaurFactoryTest extends TestCase
10  {
... lines 11 - 46
47      public function testItGrowsADinosaurFromSpecification()
48      {
... lines 49 - 50
51          $this->assertGreaterThanOrEqual(Dinosaur::LARGE, $dinosaur->getLength());
52
53          $this->assertTrue($dinosaur->isCarnivorous(), 'Diets do not match');
54      }
55  }
```

Ok! That's a nice-looking test! Before I even try it, let's *start* the code. In `DinosaurFactory`, add public function `growFromSpecification()` with a string argument. This will return a `Dinosaur`.

```
27 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
16  }
... lines 17 - 25
26 }
```

Now, our test looks happier... except apparently we do *not* have a `Dinosaur::isCarnivorous()` method. That's awesome! Another example of *not* adding a method until we need it... which is now. At the bottom of `Dinosaur`, add public function `isCarnivorous()` and return the property.

```
68 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 62
63  public function isCarnivorous(): bool
64  {
65      return $this->isCarnivorous;
66  }
67 }
```

Perfect! Our test - and even some of our *code* - is ready. Run it:

```
● ● ●
$ ./vendor/bin/phpunit
```

Whoo! It fails! So... time to code, right?

Testing Many Input at Once

Well... hold on! Because... this is going to be painful! So far, we're just testing *one* string. But this is going to be a complex function... we're going to need to test *a lot* of specifications, like "small herbivore", "huge dinosaur" or maybe even "tiny, adorable, flesh-eating carnivorous dino". To do that, we're going to need to copy and paste this method, over and over again.

Unfortunately... there's no better option. Pff, kidding! Of course there is! Data providers!

Hello Data Providers

Data providers let you run the *same* test in a loop: passing different *arguments* each time.

First, create a new public function called `getSpecificationTests()`. Yep, this method does *not* start with the word `test`. That's because its job is *not* to be a test: it's to *provide* the different test cases that we want to try.

73 lines | [tests/AppBundle/Factory/DinosaurFactoryTest.php](#)

... lines 1 - 8

```
9 class DinosaurFactoryTest extends TestCase
10 {
    ... lines 11 - 62
63     public function getSpecificationTests()
64     {
        ... lines 65 - 70
71     }
72 }
```

Let's code this first: it will make more sense when you see all the pieces working together. Return an array. Then, I'll add some comments: specification, is large and is carnivorous.

Copy the specification from above. Then, inside the array, create *another* array with that string, then true and true, because we *expect* this dinosaur to be large and carnivorous. This will be the *first* test case: we want to test that this spec will create a large, carnivorous dinosaur.

73 lines | [tests/AppBundle/Factory/DinosaurFactoryTest.php](#)

... lines 1 - 8

```
9 class DinosaurFactoryTest extends TestCase
10 {
    ... lines 11 - 62
63     public function getSpecificationTests()
64     {
65         return [
66             // specification, is large, is carnivorous
67             ['large carnivorous dinosaur', true, true],
        ... lines 68 - 69
70         ];
71     }
72 }
```

Add another item to the array with a completely ridiculous spec: give me all the cookies!!!. This time, use false and false. Management told us that if they say something crazy, the DinosaurFactory should *default* to creating small, herbivore dinosaurs... which seems like a pretty safe idea.

Add one last test case: large herbivore, which we expect to be large true and carnivorous false.

73 lines | [tests/AppBundle/Factory/DinosaurFactoryTest.php](#)

... lines 1 - 8

```
9 class DinosaurFactoryTest extends TestCase
10 {
    ... lines 11 - 62
63     public function getSpecificationTests()
64     {
65         return [
        ... lines 66 - 67
68             ['give me all the cookies!!!', false, false],
69             ['large herbivore', true, false],
70         ];
71     }
72 }
```

[Hooking up the Data Provider](#)

Ok! We're not done yet... but once we are, PHPUnit will call our test method one time for *each* item in the array... so three

times in total. On each call, it will pass the values as the arguments. So add three arguments: `$spec`, `$expectedIsLarge` and `$expectedIsCarnivorous`.

```
73 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9   class DinosaurFactoryTest extends TestCase
10  {
... lines 11 - 49
50     public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsLarge, bool $expectedIsCarnivorous)
51     {
... lines 52 - 60
61     }
... lines 62 - 71
72 }
```

Pass the dynamic spec to `growFromSpecification()`. The `greaterThanOrEqualTo` assert will now need to *vary*, depending on whether or not we're expecting a large or small dinosaur. Add if `$expectedIsLarge`. In that case, use the same assert. Else, copy that line, but use `assertLessThan()`.

```
73 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9   class DinosaurFactoryTest extends TestCase
10  {
... lines 11 - 49
50     public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsLarge, bool $expectedIsCarnivorous)
51     {
... lines 52 - 53
54         if ($expectedIsLarge) {
55             $this->assertGreaterThanOrEqualTo(Dinosaur::LARGE, $dinosaur->getLength());
56         } else {
57             $this->assertLessThan(Dinosaur::LARGE, $dinosaur->getLength());
58         }
... lines 59 - 60
61     }
... lines 62 - 71
72 }
```

Finally, use `assertSame()` at the bottom to assert that `$expectedIsCarnivorous()` matches `$dinosaur->isCarnivorous()`.

```
73 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9   class DinosaurFactoryTest extends TestCase
10  {
... lines 11 - 49
50     public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsLarge, bool $expectedIsCarnivorous)
51     {
... lines 52 - 59
60         $this->assertSame($expectedIsCarnivorous, $dinosaur->isCarnivorous(), 'Diets do not match');
61     }
... lines 62 - 71
72 }
```

Phew! Ok, the test will *not* pass yet... but I like errors! Try the tests:

```
● ● ●
$ ./vendor/bin/phpunit
```


Too few arguments to testItGrowsADinosaurFromASpecification(): 0 passed and 3 expected.

That makes sense: normally, you are *not* allowed to have *any* arguments to your test methods. But with a data provider, you can! We just need to hook it up. How? Above the test method, add `@dataProvider` `getSpecificationTests`.

```
73 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
... lines 11 - 46
47  /**
48   * @dataProvider getSpecificationTests
49   */
50  public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsLarge, bool $expectedIsCarnivorous)
51  {
... lines 52 - 60
61  }
... lines 62 - 71
72 }
```

Woo! Now, it will call the test *three* times: once for the first data set, passing these as the first, second and third arguments, then a second time, and a third. Check it out:

```
● ● ●
$ ./vendor/bin/phpunit
```

Yay! Failures! But, 3 failures! And they're labeled as with data set #0, #1 and #2... because us programmers like to count from 0.

This is *awesome*. Oh, and if you *want* to, you can assign an array key to any test, like default response for the second test. *Now* when you run the test, the second failure is called default response. I actually do this a lot - it helps figure out *which* test is actually failing.

Now that we have our three test cases, it's time to move on to Step 2 of TDD and make these pass one-by-one.

Chapter 8: Coding, Adding Features, Refactoring

Ok, let's code! And remember: don't over-think things: just focus on getting each test to pass. Let's start with the *second* test, the default values.

Attacking Test #1

Inside DinosaurFactory, I'll paste a few default values: we'll use `$codeName` as the genus, because these are *experimental* dinosaurs, set the `$length` to be a small dinosaur, and create leaf-eating friends.

```
35 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
16      // defaults
17      $codeName = 'InG-' . random_int(1, 99999);
18      $length = random_int(1, Dinosaur::LARGE - 1);
19      $isCarnivorous = false;
... lines 20 - 23
24  }
... lines 25 - 33
34  }
```

Yep, with these values, our second test should be happy. Finish the function: `$dinosaur = $this->createDinosaur()` with `$codeName`, `$isCarnivorous` and `$length`. Then, return `$dinosaur`. Oh... and it doesn't really matter... but let's move this function up: I like to have my *public* functions *above* private ones.

```
35 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
... lines 16 - 20
21      $dinosaur = $this->createDinosaur($codeName, $isCarnivorous, $length);
22
23      return $dinosaur;
24  }
... lines 25 - 33
34  }
```

Ok, that should be enough to get *one* test to pass. Run 'em:

```
● ● ●
$ ./vendor/bin/phpunit
```

Yes! Failure... *dot*... failure.

Attacking Test #2

Keep going! Let's work on the *last* test next: if the spec has the word *large* in it, it should be a large dinosaur. That's easy enough: inside the method: use `strpos` to check if the `$specification` contains `large`. Because if it *does*... we need a bigger length! Generate a random number between the `LARGE` constant and 100... which would be a *horribly* big dinosaur.

```
39 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
... lines 16 - 20
21      if (strpos($specification, 'large') !== false) {
22          $length = random_int(Dinosaur::LARGE, 100);
23      }
... lines 24 - 27
28  }
... lines 29 - 37
38  }
```

And *just* like that, another test passes!

Attack Test #3

This is fun! It's like, every time I write a line of code, Sebastian Bergmann is *personally* giving me a high five!

Ok, the *last* test is one where the spec includes the word `carnivorous`. What's the *quickest* way to get this test to pass? Just copy the if statement, paste it, change the string to `carnivorous` and set `isCarnivorous` to `true`.

```
43 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
... lines 16 - 24
25      if (strpos($specification, 'carnivorous') !== false) {
26          $isCarnivorous = true;
27      }
... lines 28 - 31
32  }
... lines 33 - 41
42  }
```

And now... thanks to the *power* of TDD... they *all* pass! That felt *great*.

We Want HUGE Dinosaurs

And management already *loves* this feature. But... they don't think the dinosaurs are *big* enough. Now, they want to use the word "huge" to grow mouth-gaping dinosaurs! They've gone mad!

No problem! Thanks to the power of data providers, we can just add more test cases! Or... if you feel like this method is already doing enough, you can create another test. Let's do that: `testItGrowsAHugeDinosaur()` with *only* a `$specification` argument. Grow the dino with `$dinosaur = $this->factory->growFromSpecification()`. Then, check to make sure it's *huge* with `$this->assertGreaterThanOrEqual()`.

```

90 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
... lines 11 - 75
76  public function testItGrowsAHugeDinosaur(string $specification)
77  {
78      $dinosaur = $this->factory->growFromSpecification($specification);
79
80      $this->assertGreaterThanOrEqual(Dinosaur::HUGE, $dinosaur->getLength());
81  }
... lines 82 - 90

```

Oh, but we need to define what *huge* means. Back in Dinosaur, add const HUGE = 30. And management decided to make the large dinosaurs a bit smaller - set LARGE to 10.

```

69 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
13     const LARGE = 10;
14     const HUGE = 30;
... lines 15 - 67
68 }

```

Use the constant in the test and compare it with `$dinosaur->getLength()`.

```

90 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
... lines 11 - 75
76  public function testItGrowsAHugeDinosaur(string $specification)
77  {
... lines 78 - 79
80      $this->assertGreaterThanOrEqual(Dinosaur::HUGE, $dinosaur->getLength());
81  }
... lines 82 - 90

```

[Huge Data Provider](#)

With the test function done, create the data provider: `getHugeDinosaurSpecTests()`. Just like before, make this return an array. Each individual test case will *also* be an array like last time, but now with only one item inside. Test for 'huge dinosaur', then also huge dino, just the word huge and, of course, OMG and... the scream Emoji!

```

94 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
... lines 11 - 82
83  public function getHugeDinosaurSpecTests()
84  {
85      return [
86          ['huge dinosaur'],
87          ['huge dino'],
88          ['huge'],
89          ['OMG'],
90          ['💩'],
91      ];
92  }
93  }

```

Back on the test method, connect it to the provider: @dataProvider getHugeDinosaurSpecTests.

```

90 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 8
9  class DinosaurFactoryTest extends TestCase
10 {
... lines 11 - 72
73  /**
74   * @dataProvider getHugeDinosaurSpecTests
75   */
76  public function testItGrowsAHugeDinosaur(string $specification)
... lines 77 - 90

```

Ok, let's watch some tests fail! Go Sebastian go!

```

$ ./vendor/bin/phpunit

```

Beautiful failures! Five new test cases and five new failures. Time to code!

In DinosaurFactory, this method is going to start getting ugly... but I don't care! Remember, our main job is to get the tests to pass, *not* to write really fancy code. TDD helps keep us focused.

First, update the large if statement to make sure it creates large, but not HUGE dinosaurs. We could have updated our test *first* before making this change.

```

47 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
... lines 16 - 24
25      if (strpos($specification, 'large') !== false) {
26          $length = random_int(Dinosaur::LARGE, Dinosaur::HUGE - 1);
27      }
... lines 28 - 35
36  }
... lines 37 - 45
46  }

```

Now, let's handle the HUGE dinos. Copy the large if statement, change the search text to huge, and generate a length between HUGE and 100.

```

47 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
... lines 16 - 20
21      if (strpos($specification, 'huge') !== false) {
22          $length = random_int(Dinosaur::HUGE, 100);
23      }
... lines 24 - 35
36  }
... lines 37 - 45
46  }

```

Run the tests!

```
$ ./vendor/bin/phpunit
```

Easy! 3 of the 5 already pass: just OMG and the screaming Emoji left! Copy the huge if statement and paste two more times. Use OMG on the first and the screaming Emoji for the second.

```

29 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
... lines 16 - 24
25      if (strpos($specification, 'OMG') !== false) {
26          $length = random_int(Dinosaur::HUGE, 100);
27      }
... line 28
29      if (strpos($specification, '

```

I know... there's so much duplication! It's so ugly. But... I don't care! I love it because the tests *do* pass!

Refactoring our Ugly Code

And *that* means we've reached step 3 of TDD: refactor! I don't *actually* love ugly code - it's just that it wasn't time to worry about it yet. TDD helps you focus on writing your business logic correctly *first*, and *then* on improving the code.

So let's make this better. Actually, if you downloaded the course code, then you should have a tutorial/ directory with a DinosaurFactory.php file inside. Copy the private function from that file, find *our* DinosaurFactory, and paste at the bottom.

```
64 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 38
39  private function getLengthFromSpecification(string $specification): int
40  {
41      $availableLengths = [
42          'huge' => ['min' => Dinosaur::HUGE, 'max' => 100],
43          'omg' => ['min' => Dinosaur::HUGE, 'max' => 100],
44          '🦖' => ['min' => Dinosaur::HUGE, 'max' => 100],
45          'large' => ['min' => Dinosaur::LARGE, 'max' => Dinosaur::HUGE - 1],
46      ];
47      $minLength = 1;
48      $maxLength = Dinosaur::LARGE - 1;
49
50      foreach (explode(' ', $specification) as $keyword) {
51          $keyword = strtolower($keyword);
52
53          if (array_key_exists($keyword, $availableLengths)) {
54              $minLength = $availableLengths[$keyword]['min'];
55              $maxLength = $availableLengths[$keyword]['max'];
56
57              break;
58          }
59      }
60
61      return random_int($minLength, $maxLength);
62  }
63 }
```

This is still a bit complex, but it removes the duplication and makes the length calculation more systematic. Copy the method name, scroll up, delete *all* that ugly length logic... and just say `$length = $this->getLengthFromSpecification($specification)`.

```
44 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 6
7  class DinosaurFactory
8  {
... lines 9 - 13
14  public function growFromSpecification(string $specification): Dinosaur
15  {
... lines 16 - 17
18      $length = $this->getLengthFromSpecification($specification);
... lines 19 - 27
28  }
... lines 29 - 44
```

My new code *probably* doesn't contain any bugs... but you should *totally* not trust me! I mess up all the time! Just run the tests.

A terminal window with a dark gray title bar containing three small circular window control buttons (red, yellow, green). The terminal's main area is black, and the prompt '\$./vendor/bin/phpunit' is displayed in a light gray monospaced font.

```
$ ./vendor/bin/phpunit
```

Ha! It works! And you doubted me....

Next! What if you need to test that a method throws an *exception* under certain conditions? Like... if you try to put a T-Rex in the same enclosure as a nice, friendly Brontosaurus. Let's find out!

Chapter 9: Handling Object Dependencies

Now that we're building *all* these dinosaurs... we need a place to keep them! Right now they're running free! Terrorizing the guests! Eating all the ice cream! We need an Enclosure class that will hold a collection of dinosaurs.

You guys know the drill, start with the test! Create EnclosureTest.

We don't want any surprise dinosaurs inside!

Create the new Enclosure() and then check that `$this->assertCount(0)` matches `$enclosure->getDinosaurs()`.

```
17 lines | tests/AppBundle/Entity/EnclosureTest.php
... lines 1 - 7
8  class EnclosureTest extends TestCase
9  {
10     public function testItHasNoDinosaursByDefault()
11     {
12         $enclosure = new Enclosure();
13
14         $this->assertCount(0, $enclosure->getDinosaurs());
15     }
16 }
```

Ok, good start! Next, inside Entity, create Enclosure. This will eventually be a Doctrine entity, but don't worry about the annotations yet. Add a private `$dinosaurs` property. And, like normal, add public function `__construct()` so that we can initialize that to a new `ArrayCollection`.

```
26 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 8
9  class Enclosure
10 {
... lines 11 - 13
14     private $dinosaurs;
... line 15
16     public function __construct()
17     {
18         $this->dinosaurs = new ArrayCollection();
19     }
... lines 20 - 24
25 }
```

Back on the property, I'll add `@var Collection`. That's the interface that `ArrayCollection` implements.

```
26 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 8
9  class Enclosure
10 {
11     /**
12      * @var Collection
13      */
14     private $dinosaurs;
... lines 15 - 24
25 }
```

Now that the class exists, go back to the test and add the use statement. Oh... and PhpStorm doesn't like my `assertCount()` method... because I forgot to extend `TestCase`!

```
17 lines | tests/AppBundle/Entity/EnclosureTest.php
... lines 1 - 4
5  use AppBundle\Entity\Enclosure;
... lines 6 - 7
8  class EnclosureTest extends TestCase
9  {
... lines 10 - 15
16 }
```

If we run the test now, it - of course - fails:

```
$ ./vendor/bin/phpunit
```

In `Enclosure`, finish the code by adding `getDinosaurs()`, which should return a `Collection`. Summon the tests!

```
26 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 8
9  class Enclosure
10 {
... lines 11 - 20
21     public function getDinosaurs(): Collection
22     {
23         return $this->dinosaurs;
24     }
25 }
```

```
$ ./vendor/bin/phpunit
```

We are green! I know, this is simple so far... but stay tuned.

[Adding the Annotations](#)

Before we keep going, since the tests are green, let's add the missing Doctrine annotations. With my cursor inside `Enclosure`, I'll go to the `Code->Generate` menu - or `Command+N` on a mac - and select "ORM Class". That's just a shortcut to add the annotations above the class.

```
31 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 8
9  /**
10   * @ORM\Entity
11   * @ORM\Table(name="enclosures")
12   */
13  class Enclosure
14  {
... lines 15 - 29
30 }
```

Now, above the `$dinosaurs` property, use `@ORM\OneToMany` with `targetEntity="Dinosaur"`, `mappedBy="enclosure"` - we'll add that property in a moment - and `cascade={"persist"}`.

```

31 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 8
9  /**
10 * @ORM\Entity
11 * @ORM\Table(name="enclosures")
12 */
13 class Enclosure
14 {
15     /**
16      * @var Collection
17      * @ORM\OneToMany(targetEntity="AppBundle\Entity\Dinosaur", mappedBy="enclosure", cascade={"persist"})
18      */
19     private $dinosaurs;
... lines 20 - 29
30 }

```

In Dinosaur, add the other side: private \$enclosure with @ORMManyToOne. Point *back* to the Enclosure class with inversedBy="dinosaurs".

```

75 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 32
33     /**
34      * @var Enclosure
35      * @ORM\ManyToOne(targetEntity="AppBundle\Entity\Enclosure", inversedBy="dinosaurs")
36      */
37     private $enclosure;
... lines 38 - 73
74 }

```

That should *not* have broken anything... but run the tests to be sure!

```
$ ./vendor/bin/phpunit
```

Dependent Objects

Testing that the enclosure starts *empty* is great... but we need a way to add dinosaurs! Create a new method: testItAddsDinosaurs(). Then, instantiate a new Enclosure() object.

```

28 lines | tests/AppBundle/Entity/EnclosureTest.php
... lines 1 - 8
9  class EnclosureTest extends TestCase
10 {
... lines 11 - 17
18     public function testItAddsDinosaurs()
19     {
20         $enclosure = new Enclosure();
... lines 21 - 25
26     }
27 }

```

Design phase! How should we allow dinosaurs to be added to an Enclosure? Maybe... an addDinosaur() method. Brilliant! \$enclosure->addDinosaur(new Dinosaur()).

```

28 lines | tests/AppBundle/Entity/EnclosureTest.php
... lines 1 - 8
9   class EnclosureTest extends TestCase
10  {
... lines 11 - 17
18      public function testItAddsDinosaurs()
19      {
... lines 20 - 21
22          $enclosure->addDinosaur(new Dinosaur());
... lines 23 - 25
26      }
27  }

```

And *this* is where things get interesting. For the *first* time, in order to test one class - Enclosure - we need an object of a *different* class - Dinosaur. A unit test is *supposed* to test *one* class in *complete* isolation from all other classes. We want to test the logic of *Enclosure*, not Dinosaur.

This is why *mocking* exists. With mocking, instead of instantiating and passing *real* objects - like a real Dinosaur object - you create a "fake" object that *looks* like a Dinosaur, but isn't. As you'll see in a few minutes, a mock object gives you a lot of control.

Mock the Dinosaur?

So... should we *mock* this Dinosaur object? Actually... no. I know we haven't even *seen* mocking yet, but let me give you a general rule to follow:

When you're testing an object (like Enclosure) and this requires you to create an object of a *different* class (like Dinosaur), *only* mock this object if it is a *service*. Mock *services*, but don't mock simple model objects.

Let me say it a different way: if you're organizing your code well, then all classes will fall into one of two types. The first type - a *model* class - is a class whose job is basically to hold data... but not do much work. Our entities are model classes. The second type - a *service* class - is a class whose main job is to do *work*, but it doesn't hold much data, other than maybe some configuration. DinosaurFactory is a service class.

As a rule, you *will* want to mock *service* classes, but you do *not* need to mock *model* classes. Why not? Well, you *can*... but usually it's overkill. Since model classes tend to be simple and just hold data, it's easy enough to create those objects and set their data to whatever you want.

If this does not make sense yet, don't worry. We're going to talk about mocking *very* soon.

Let's add one more dinosaur to the enclosure. And then check that `$this->assertCount(2)` equals `$enclosure->getDinosaurs()`.

```

28 lines | tests/AppBundle/Entity/EnclosureTest.php
... lines 1 - 8
9   class EnclosureTest extends TestCase
10  {
... lines 11 - 17
18      public function testItAddsDinosaurs()
19      {
... lines 20 - 21
22          $enclosure->addDinosaur(new Dinosaur());
23          $enclosure->addDinosaur(new Dinosaur());
24
25          $this->assertCount(2, $enclosure->getDinosaurs());
26      }
27  }

```

Try the test!

```
$ ./vendor/bin/phpunit
```

Of course, it *fails* due to the missing method. Open `Enclosure` and create public function `addDinosaur()` with a `Dinosaur` argument. When you finish, try the tests again:

```
36 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 12
13 class Enclosure
14 {
... lines 15 - 30
31     public function addDinosaur(Dinosaur $dinosaur)
32     {
33         $this->dinosaurs[] = $dinosaur;
34     }
35 }
```

```
$ ./vendor/bin/phpunit
```

Oh, and one last thing! Instead of `$this->assertCount(0)`, you can use `$this->assertEmpty()`... which just sounds cooler. It works the same.

```
28 lines | tests/AppBundle/Entity/EnclosureTest.php
... lines 1 - 8
9 class EnclosureTest extends TestCase
10 {
11     public function testItHasNoDinosaursByDefault()
12     {
... lines 13 - 14
15         $this->assertEmpty($enclosure->getDinosaurs());
16     }
... lines 17 - 26
27 }
```

Ok, *now* let's talk exceptions!

Chapter 10: Testing Exceptions

The dinosaur enclosures are looking *great*. There's just one *minor* problem: people keep accidentally putting nice, gentle veggie-eating dinosaurs into the same enclosure as flesh-eating carnivores. The result is... well... *expensive*, and there's a lot of cleanup too.

We need to prevent the meat eaters and the veggie eaters from being mixed inside the same enclosure. In other words, if somebody tries to do this, we need to throw an exception!

The Custom Exception Class

It's optional, but let's create a custom exception class for this: `NotABuffetException`. I'll even give this a default message: people need to understand how horrible this is!

9 lines | `src/AppBundle/Exception/NotABuffetException.php`

```
... lines 1 - 2
3 namespace AppBundle\Exception;
... line 4
5 class NotABuffetException extends \Exception
6 {
7     protected $message = 'Please do not mix the carnivorous and non-carnivorous dinosaurs. It will be a massacre!';
8 }
```

Making sure that this exception is thrown at the right time is *critical* to business. So let's write a test: `testItDoesNotAllowCarnivorousDinosaursToMixWithHerbivores`.

Inside the method, create this terrifying situation: `$enclosure = new Enclosure()` and `$enclosure->addDinosaur(new Dinosaur())`. By default, dinosaurs are *non-carnivorous*. So now, let's add a predator: `new Dinosaur('Velociraptor')` and `true` for the `isCarnivorous` argument.

40 lines | `tests/AppBundle/Entity/EnclosureTest.php`

```
... lines 1 - 9
10 class EnclosureTest extends TestCase
11 {
... lines 12 - 28
29     public function testItDoesNotAllowCarnivorousDinosToMixWithHerbivores()
30     {
31         $enclosure = new Enclosure();
32
33         $enclosure->addDinosaur(new Dinosaur());
... lines 34 - 36
37         $enclosure->addDinosaur(new Dinosaur('Velociraptor', true));
38     }
39 }
```

Expecting an Exception

At this point, an exception *should* be thrown. So... how can we test for that? By telling PHPUnit to *expect* an exception with... well... `$this->expectException()` and then the exception class: `NotABuffetException::class`. Make sure you add this *before* calling the final code.

```

40 lines | tests/AppBundle/Entity/EnclosureTest.php
... lines 1 - 9
10 class EnclosureTest extends TestCase
11 {
... lines 12 - 28
29     public function testItDoesNotAllowCarnivorousDinosToMixWithHerbivores()
30     {
... lines 31 - 34
35         $this->expectException(NotABuffetException::class);
... lines 36 - 37
38     }
39 }

```

If we've done our work correctly, this should fail. Try the test!

```

$ ./vendor/bin/phpunit

```

Yes! Failed asserting that exception of type NotABuffetException is thrown.

Awesome! Let's go throw that exception! Inside Enclosure, at the bottom, add a new private function called canAddDinosaur with a Dinosaur argument. This will return a bool.

```

47 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 13
14 class Enclosure
15 {
... lines 16 - 40
41     private function canAddDinosaur(Dinosaur $dinosaur): bool
42     {
... lines 43 - 44
45     }
46 }

```

Here's some simple logic: return count(\$this->dinosaurs) === 0. So, if the enclosure is empty, then it's *definitely* ok to add a dinosaur. Or, check to see if \$this->dinosaurs->first()->isCarnivorous() === \$dinosaur->isCarnivorous(). If they match, we're good!

```

47 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 13
14 class Enclosure
15 {
... lines 16 - 40
41     private function canAddDinosaur(Dinosaur $dinosaur): bool
42     {
43         return count($this->dinosaurs) === 0
44             || $this->dinosaurs->first()->isCarnivorous() === $dinosaur->isCarnivorous();
45     }
46 }

```

Back in addDinosaur(), if not \$this->canAddDinosaur(). Throw the exception! Oh wait... make sure the class extends \Exception. My bad!

```

9 lines | src/AppBundle/Exception/NotABuffetException.php
... lines 1 - 4
5 class NotABuffetException extends \Exception
6 {
... line 7
8 }

```

Now throw that exception!

```

47 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 13
14 class Enclosure
15 {
... lines 16 - 31
32 public function addDinosaur(Dinosaur $dinosaur)
33 {
34     if (!$this->canAddDinosaur($dinosaur)) {
35         throw new NotABuffetException();
36     }
... lines 37 - 38
39 }
... lines 40 - 45
46 }

```

Check the tests!

```

$ ./vendor/bin/phpunit

```

Woo! We got it!

[expect Exceptions via Annotations](#)

There's one other way to test for exceptions. It's really the same, but looks fancier. Copy the test method and rename it so we can test for the *opposite* condition: `testItDoesNotAllowToAddNonCarnivorousDinosaursToCarnivorousEnclosure`. Wow that's a long name!

```

51 lines | tests/AppBundle/Entity/EnclosureTest.php
... lines 1 - 9
10 class EnclosureTest extends TestCase
11 {
... lines 12 - 42
43 public function testItDoesNotAllowToAddNonCarnivorousDinosaursToCarnivorousEnclosure()
44 {
... lines 45 - 48
49 }
50 }

```

Add the *Velociraptor* *first* and then remove `expectException`. Instead, add an annotation: `@expectedException` followed by the full class. PhpStorm puts the short name... so go copy the use statement and put it down here. Try it!

51 lines | [tests/AppBundle/Entity/EnclosureTest.php](#)

... lines 1 - 9

10 class EnclosureTest extends TestCase

11 {

... lines 12 - 39

40 /**

41 * @expectedException \AppBundle\Exception\NotABuffetException

42 */

43 public function testItDoesNotAllowToAddNonCarnivorousDinosaursToCarnivorousEnclosure()

44 {

45 \$enclosure = new Enclosure();

46

47 \$enclosure->addDinosaur(new Dinosaur('Velociraptor', true));

48 \$enclosure->addDinosaur(new Dinosaur());

49 }

50 }



\$./vendor/bin/phpunit

Yes! One more test passing.

I want to go through *one* more example next... and also add some security to the enclosures. Our guests have been terrorized enough.

Chapter 11: Exceptions Part 2: Adding Fence Security

Dinosaurs, check! Enclosures, check! But... we forgot to add *security* to the enclosures! Ya know, like *electric* fences! Dang it, I *knew* we forgot something. The dinosaurs have been escaping their enclosure and... of course, terrorizing the guests. The investors are not going to like this...

Inside Enclosure, add a new securities property: this will be a collection of Security objects - like "fence" security or "watch tower". We'll create that class in a minute. Anyways, if this collection is *empty*, there's no security! So we *cannot* let people put dinosaurs here.

```
49 lines | src/AppBundle/Entity/Enclosure.php
```

```
... lines 1 - 13
14 class Enclosure
15 {
... lines 16 - 21
22     private $securities;
... lines 23 - 47
48 }
```

Let's create another custom exception class: DinosaursAreRunningRampantException. This time, make sure it extends \Exception.

```
8 lines | src/AppBundle/Exception/DinosaursAreRunningRampantException.php
```

```
... lines 1 - 4
5 class DinosaursAreRunningRampantException extends \Exception
6 {
7 }
```

Perfect!

Testing the Dinosaurs don't Run Ramptant

Inside EnclosureTest, add a new method: testItDoesNotAllowToAddDinosToUnsecureEnclosures. And yea... this is pretty simple: just create the new Enclosure(), and then add the dinosaur. But first, this time, I want to test for the exception *class* and exception message. You can do *both* via annotations, or right here with `$this->expectException(DinosaursAreRunningRamptantException::class)` and `$this->expectExceptionMessage('Are you craaazy?!?')`.

Below that, add a new Dinosaur().

62 lines | [tests/AppBundle/Entity/EnclosureTest.php](#)

... lines 1 - 10

```
11 class EnclosureTest extends TestCase
```

```
12 {
```

... lines 13 - 51

```
52     public function testItDoesNotAllowToAddDinosToUnsecureEnclosures()
```

```
53     {
```

```
54         $enclosure = new Enclosure();
```

```
55
```

```
56         $this->expectException(DinosaursAreRunningRampantException::class);
```

```
57         $this->expectExceptionMessage('Are you craaazy?!?');
```

```
58
```

```
59         $enclosure->addDinosaur(new Dinosaur());
```

```
60     }
```

```
61 }
```

Nice! Except... yea... this is going to make *all* of our tests fail: none of those Enclosures have security. Let's worry about that later: focus on *this* test.

In fact, copy the method name and find your terminal. To avoid noise, I want to run *only* this *one* test. You can do that with `./vendor/bin/phpunit --filter` and then the method:



```
$ ./vendor/bin/phpunit --filter testItDoesNotAllowToAddDinosToUnsecureEnclosures
```

Awesome! One test and one failure. We'll talk more about `--filter` soon!

[Adding the Security Class](#)

Ok, step 2 of TDD: code! First, we need a Security class. If you downloaded the course code, you should have a `tutorial/` directory with a Security class inside. Paste that into our Entity directory.

47 lines | [src/AppBundle/Entity/Security.php](#)

... lines 1 - 2

```
3 namespace AppBundle\Entity;
... line 4
5 use Doctrine\ORM\Mapping as ORM;
... line 6
7 /**
8  * @ORM\Entity
9  * @ORM\Table(name="securities")
10 */
11 class Security
12 {
13     /**
14      * @ORM\Id
15      * @ORM\GeneratedValue(strategy="AUTO")
16      * @ORM\Column(type="integer")
17      */
18     private $id;
19
20     /**
21      * @ORM\Column(type="string")
22      */
23     private $name;
24
25     /**
26      * @ORM\Column(type="boolean")
27      */
28     private $isActive;
29
30     /**
31      * @ORM\ManyToOne(targetEntity="AppBundle\Entity\Enclosure", inversedBy="securities")
32      */
33     private $enclosure;
34
35     public function __construct(string $name, bool $isActive, Enclosure $enclosure)
36     {
37         $this->name = $name;
38         $this->isActive = $isActive;
39         $this->enclosure = $enclosure;
40     }
41
42     public function getIsActive(): bool
43     {
44         return $this->isActive;
45     }
46 }
```

It's pretty simple: it has a name, an isActive boolean and a reference to the Enclosure it's attached to.

Speaking of Enclosure, initialize its \$securities property to a new ArrayCollection. Oh, and on the property, add @var Collection|Security[].

```

79 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 14
15 class Enclosure
16 {
... lines 17 - 22
23 /**
24  * @var Collection
... line 25
26 */
27 private $securities;
28
29 public function __construct(bool $withBasicSecurity = false)
30 {
... line 31
32     $this->securities = new ArrayCollection();
... lines 33 - 36
37 }
... lines 38 - 77
78 }

```

Really, this will be a Collection instance. But the `Security[]` part tells our editor that this is a collection of *Security* objects. And that will give *us* better auto-completion. Which we can only enjoy if we get this dino security going, so let's get to it!

Throwing the Exception

Down in `addDinosaur()`, we need to know if this Enclosure has at least *one* active security. Add a method to help with that: `public function isSecurityActive()`. I'm making this public *only* because I already know I'm going to use it later outside of this class.

Set this to return a bool and then loop! Iterate over `$this->securities` as `$security`. And if `$security->getIsActive()`, return true. If there are *no* active securities, run for your life! And also return false at the bottom.

```

79 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 14
15 class Enclosure
16 {
... lines 17 - 67
68 public function isSecurityActive(): bool
69 {
70     foreach ($this->securities as $security) {
71         if ($security->getIsActive()) {
72             return true;
73         }
74     }
75
76     return false;
77 }
78 }

```

Finish things in `addDinosaur()`: if not `$this->isSecurityActive()`, throw a new `DinosaursAreRunningRampantException()`. And remember, we're checking for an exact message: so use the string from the test.

```

79 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 14
15 class Enclosure
16 {
... lines 17 - 43
44     public function addDinosaur(Dinosaur $dinosaur)
45     {
46         if (!$this->isSecurityActive()) {
47             throw new DinosaurusAreRunningRampantException('Are you craazy?!?');
48         }
... lines 49 - 54
55     }
... lines 56 - 77
78 }

```

Ok, I think we're done! Go tests go!

```
$ ./vendor/bin/phpunit --filter testItDoesNotAllowToAddDinosToUnsecureEnclosures
```

Yes! It's now *impossible* to add a Dinosaur to an Enclosure... unless there's some security to keep it inside.

Adding Annotations

We've reached the *third* step of TDD once again: refactor. Actually, I don't need to refactor, but now is a *great* time to add the missing Doctrine annotations. Above the \$securities property, add `@ORM\OneToMany()` with `targetEntity="Security"` and `mappedBy="enclosure"`. enclosure is the name of the property on the other side of the relation. Finish it with `cascade={"persist"}`.

```

79 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 14
15 class Enclosure
16 {
... lines 17 - 22
23     /**
... line 24
25     * @ORM\OneToMany(targetEntity="AppBundle\Entity\Security", mappedBy="enclosure", cascade={"persist"})
26     */
27     private $securities;
... lines 28 - 77
78 }

```

Ok, this *one* test still passes... but what about the rest? Run them:

```
$ ./vendor/bin/phpunit
```

Ah! We have so many DinosaurusAreRunningRampantException errors! Yep, we knew this was coming: our *existing* tests need security.

To make this easy, inside Enclosure, add a bool `$withBasicSecurity` argument. Then, if this is true, let's add some basic security! `$this->addSecurity()` - we'll create this method next - new Security('Fence', true) - for `isActive` - and then `$this` for the Enclosure.

```

79 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 14
15 class Enclosure
16 {
... lines 17 - 28
29     public function __construct(bool $withBasicSecurity = false)
30     {
... lines 31 - 33
34         if ($withBasicSecurity) {
35             $this->addSecurity(new Security('Fence', true, $this));
36         }
37     }
... lines 38 - 77
78 }

```

Add the missing method public function addSecurity(), append the securities array and... we're done!

```

79 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 14
15 class Enclosure
16 {
... lines 17 - 56
57     public function addSecurity(Security $security)
58     {
59         $this->securities[] = $security;
60     }
... lines 61 - 77
78 }

```

Inside EnclosureTest, the first test method does *not* need security: it never adds any dinosaurs. But the next three do: pass true, true and true.

62 lines | [tests/AppBundle/Entity/EnclosureTest.php](#)

... lines 1 - 10

```
11 class EnclosureTest extends TestCase
12 {
```

... lines 13 - 19

```
20     public function testItAddsDinosaurs()
21     {
22         $enclosure = new Enclosure(true);
```

... lines 23 - 27

```
28     }
```

... line 29

```
30     public function testItDoesNotAllowCarnivorousDinosToMixWithHerbivores()
31     {
32         $enclosure = new Enclosure(true);
```

... lines 33 - 38

```
39     }
```

... lines 40 - 43

```
44     public function testItDoesNotAllowToAddNonCarnivorousDinosaursToCarnivorousEnclosure()
45     {
46         $enclosure = new Enclosure(true);
```

... lines 47 - 49

```
50     }
```

... lines 51 - 60

```
61 }
```

Let's do the tests!



```
$ ./vendor/bin/phpunit
```

Yes! This is *amazing*! We just created a dinosaur park *with* security. What a novel idea!

Chapter 12: Refactoring & Dependency Injection

In DinosaurFactory, we made a fancy private function `getLengthFromSpecification`. It's a *great* function. So great, that now, I want to be able to use it from *outside* this class.

To do that... and of course... to help us get to mocking, let's refactor this method into its own class. Create a new Service directory in AppBundle. And then a new class called `DinosaurLengthDeterminator`. That's a fun name.

Copy `getLengthFromSpecification()`, remove it, and paste it here. Make the method public and re-type Dinosaur to get the use statement.

```
34 lines | src/AppBundle/Service/DinosaurLengthDeterminator.php
... lines 1 - 4
5 use AppBundle\Entity\Dinosaur;
... line 6
7 class DinosaurLengthDeterminator
8 {
9     public function getLengthFromSpecification(string $specification): int
10    {
11        $availableLengths = [
12            'huge' => ['min' => Dinosaur::HUGE, 'max' => 100],
13            'omg' => ['min' => Dinosaur::HUGE, 'max' => 100],
14            '🦖' => ['min' => Dinosaur::HUGE, 'max' => 100],
15            'large' => ['min' => Dinosaur::LARGE, 'max' => Dinosaur::HUGE - 1],
16        ];
17        $minLength = 1;
18        $maxLength = Dinosaur::LARGE - 1;
19
20        foreach (explode(' ', $specification) as $keyword) {
21            $keyword = strtolower($keyword);
22
23            if (array_key_exists($keyword, $availableLengths)) {
24                $minLength = $availableLengths[$keyword]['min'];
25                $maxLength = $availableLengths[$keyword]['max'];
26
27                break;
28            }
29        }
30
31        return random_int($minLength, $maxLength);
32    }
33 }
```

We *already* have a bunch of tests in `DinosaurFactoryTest` that make sure each specification string gives us the right length. In tests, create that same Service directory and a new `DinosaurLengthDeterminatorTest`. We're going to migrate the existing length tests to this class.

Add public function `testItReturnsCorrectLengthRange()` with `$spec`, `$minExpectedSize` and `$maxExpectedSize`.

34 lines | tests/AppBundle/Service/DinosaurLengthDeterminatorTest.php

... lines 1 - 8

```
9 class DinosaurLengthDeterminatorTest extends TestCase
```

```
10 {
```

... lines 11 - 13

```
14     public function testItReturnsCorrectLengthRange($spec, $minExpectedSize, $maxExpectedSize)
```

```
15     {
```

... lines 16 - 20

```
21     }
```

... lines 22 - 34

This test will be similar to the one in `DinosaurFactoryTest`, but a bit simpler: it *only* needs to test the length. Create a new determinator. And then set `$actualSize` to `$determinator->getLengthFromSpecification($spec)`.

34 lines | tests/AppBundle/Service/DinosaurLengthDeterminatorTest.php

... lines 1 - 8

```
9 class DinosaurLengthDeterminatorTest extends TestCase
```

```
10 {
```

... lines 11 - 13

```
14     public function testItReturnsCorrectLengthRange($spec, $minExpectedSize, $maxExpectedSize)
```

```
15     {
```

```
16         $determinator = new DinosaurLengthDeterminator();
```

```
17         $actualSize = $determinator->getLengthFromSpecification($spec);
```

... lines 18 - 20

```
21     }
```

... lines 22 - 34

To make sure this is within the range, add `$this->assertGreaterThanOrEqual()`. Oh wait! No auto-completion! Bah! Extend `TestCase`!

34 lines | tests/AppBundle/Service/DinosaurLengthDeterminatorTest.php

... lines 1 - 8

```
9 class DinosaurLengthDeterminatorTest extends TestCase
```

```
10 {
```

... lines 11 - 34

Now add `$this->assertGreaterThanOrEqual()` with `$minExpectedSize` and `$actualSize`. You need to read this... backwards: this asserts that `$actualSize` is greater than or equal to `$maxExpectedSize`.

Repeat that with `$this->assertLessThanOrEqual()` and `$maxExpectedSize, $actualSize`.

34 lines | tests/AppBundle/Service/DinosaurLengthDeterminatorTest.php

... lines 1 - 8

```
9 class DinosaurLengthDeterminatorTest extends TestCase
```

```
10 {
```

... lines 11 - 13

```
14     public function testItReturnsCorrectLengthRange($spec, $minExpectedSize, $maxExpectedSize)
```

```
15     {
```

... lines 16 - 18

```
19         $this->assertGreaterThanOrEqual($minExpectedSize, $actualSize);
```

```
20         $this->assertLessThanOrEqual($maxExpectedSize, $actualSize);
```

```
21     }
```

... lines 22 - 34

[Adding the Length Data Provider](#)

The *real* work is done in the data provider. Add public function `getSpecLengthTests()`. If you look at `DinosaurFactoryTest`, the

getSpecificationTests() already has great examples. Copy those, go back to the new test, and paste. We need *almost* the same thing: just change the comments to specification, min length and max length.

Then, for a large dinosaur, it should be between Dinosaur::LARGE and DINOSAUR::HUGE - 1. For a small dinosaur, the range is 0 to Dinosaur::LARGE - 1. Copy the large dino range and use that for the last one too,

```
38 lines | tests/AppBundle/Service/DinosaurLengthDeterminatorTest.php
... lines 1 - 8
9  class DinosaurLengthDeterminatorTest extends TestCase
10 {
... lines 11 - 22
23  public function getSpecLengthTests()
24  {
25      return [
26          // specification, min length, max length
27          ['large carnivorous dinosaur', Dinosaur::LARGE, Dinosaur::HUGE - 1],
28          'default response' => ['give me all the cookies!!!', 0, Dinosaur::LARGE - 1],
29          ['large herbivore', Dinosaur::LARGE, Dinosaur::HUGE - 1],
... lines 30 - 34
35  ];
36  }
37 }
```

We can *also* move the huge dinosaur tests here. Copy them, move back, and paste! This time, the range should be Dinosaur::HUGE to 100. Copy that and use it for all of them.

```
38 lines | tests/AppBundle/Service/DinosaurLengthDeterminatorTest.php
... lines 1 - 8
9  class DinosaurLengthDeterminatorTest extends TestCase
10 {
... lines 11 - 22
23  public function getSpecLengthTests()
24  {
25      return [
... lines 26 - 29
30          ['huge dinosaur', Dinosaur::HUGE, 100],
31          ['huge dino', Dinosaur::HUGE, 100],
32          ['huge', Dinosaur::HUGE, 100],
33          ['OMG', Dinosaur::HUGE, 100],
34          ['💩', Dinosaur::HUGE, 100],
35  ];
36  }
37 }
```

And *finally*, hook this all up with @dataProvider getSpecLengthTests(). I'll even fix my typo!

34 lines | tests/AppBundle/Service/DinosaurLengthDeterminatorTest.php

... lines 1 - 8

```
9 class DinosaurLengthDeterminatorTest extends TestCase
10 {
11     /**
12      * @dataProvider getSpecLengthTests
13      */
14     public function testItReturnsCorrectLengthRange($spec, $minExpectedSize, $maxExpectedSize)
15     {
16         ... lines 16 - 20
17     }
18     ... lines 22 - 34
```

Perfect! Because we deleted some code, the DinosaurFactory is temporarily broken. So let's execute *just* this test:

```
$ ./vendor/bin/phpunit tests/AppBundle/Service/DinosaurLengthDeterminatorTest.php
```

It passes!

Refactoring to use a Dependency

Time to fix the factory and get those dinosaurs growing again! The `getLengthFromSpecification()` method is gone. To use the new determinator class, use dependency injection: add public function `__construct()` with a `DinosaurLengthDeterminator` argument. I'll press alt+enter and select "Initialize fields" as a shortcut to create the property and set it below.

47 lines | src/AppBundle/Factory/DinosaurFactory.php

... lines 1 - 7

```
8 class DinosaurFactory
9 {
10     private $lengthDeterminator;
11
12     public function __construct(DinosaurLengthDeterminator $lengthDeterminator)
13     {
14         $this->lengthDeterminator = $lengthDeterminator;
15     }
16     ... lines 16 - 45
17 }
46 }
```

Back in `growFromSpecification()`, use `$this->lengthDeterminator` to call the method.

47 lines | src/AppBundle/Factory/DinosaurFactory.php

... lines 1 - 7

```
8 class DinosaurFactory
9 {
10     ... lines 10 - 21
22     public function growFromSpecification(string $specification): Dinosaur
23     {
24         ... lines 24 - 25
26         $length = $this->lengthDeterminator->getLengthFromSpecification($specification);
27         ... lines 27 - 35
36     }
37     ... lines 37 - 45
46 }
```

And... that's it! We haven't run the tests yet, but this class now *probably* works again. You need to pass in a dependency, but as long as you do that, it's all basically the same.

Run *all* the tests:

A terminal window with a dark background. The title bar is dark gray with three light gray window control buttons (minimize, maximize, close) on the left. The terminal content is a dark gray rectangle with a light gray prompt character '\$' followed by the command './vendor/bin/phpunit'.

Woh! That is a *lot* of failures - all with the same message:

Too few arguments passed to DinosaurFactory::__construct() on DinosaurFactoryTest line 18.

Scroll up to line 18. Of course: we're missing the constructor argument to the factory in our test.

For the *second* time, we have *dependent* objects. The object we're testing - DinosaurFactory - is *dependent* on another object - DinosaurLengthDeterminator. We're going to fix this with a *mock*.

Chapter 13: Mocks & Test Doubles

We're testing `DinosaurFactory`... but it *needs* a `DinosaurLengthDeterminator`! This happened before, when the `Enclosure` required a `Dinosaur` object. In *that* case, instead of *mocking* the `Dinosaur` object, we just created it. Remember the rule? If the object you need is a simple *model* object, just create it. If it's a *service*, then mock it.

Well... `DinosaurLengthDeterminator` *is* a service: it's a class that does work. So we're going to mock it. This is pretty typical for constructor arguments, which tend to be services.

By "mocking", I mean that we're going to pass an object that *looks* and smells like a `DinosaurLengthDeterminator`, but... isn't. Mocks are easier to create than the real objects, and their methods are "faked": if the *real* class queries the database or makes API calls, the mock will do nothing.

Creating the Mock Object

Let's do it! `$mockLengthDeterminator = $this->createMock(DinosaurLengthDeterminator::class)`. Pass that as the first argument.

```
69 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 16
17     public function setUp()
18     {
19         $mockLengthDeterminator = $this->createMock(DinosaurLengthDeterminator::class);
20         $this->factory = new DinosaurFactory($mockLengthDeterminator);
21     }
... lines 22 - 67
68 }
```

Before we talk about this, find your terminal, and run the tests:

```
$ ./vendor/bin/phpunit
```

Whoa! They still fail... but the error message is different: failed asserting that 0 is equal to 10 or greater than 10. Find line 81 of the test. Oh, this 0 is actually referring to `$dinosaur->getLength()`.

So... interesting... all the dinosaur lengths are being set to zero. Here's why: when you create a mock, it creates a new class in memory that extends the original, but overrides *every* method and simply returns null. That means, when we call `getLengthFromSpecification()`, it does *not* execute this code. Nope, this function in the mocked class returns null.

Well, actually, that's not *entirely* true. In this case, the length is 0. That's thanks to the return type on the method. But basically, all the methods return "nothing". A mock is a "zombie" version of the original object.

The Mock Builder

I'll hold command and click `createMock()` to see that function. Behind the scenes, mocks are created with a mock *builder*. And sometimes, you'll want to use *that* instead, because it gives you a bit more control. As you can see, by default, the constructor is skipped when creating the mock... which is pretty sweet, because you *don't* need to worry about the constructor arguments of a class.

Also, by default, *all* methods are mocked. But you can use the `setMethods()` function to only mock *some* methods.

By the way, the most common word you're going to hear for this object is a *mock*. But technically, there are a number of different terms: dummies, stubs, spies and mocks. They all mean *slightly* different things. And technically, what we've created

is a *dummy*... which is just a bit insulting!

Anyways, if you want to learn what these terms *really* mean, you can check out a series of articles written by Andrew, our course co-author at this link: <http://www.ifdattic.com/dummy-test-double-using-prophecy/>.

But for this tutorial, I'll use the word *mock* to keep things simple.

Removing the Length Tests

Ok, back to the test. By creating the mock, we fixed the original error. But now, all of the length tests are failing! But... wait a second. In a unit test, we should *only* worry about testing the logic of DinosaurFactory. And... well... DinosaurFactory no longer has *any* logic related to lengths! That calculation is done - and *tested* - in DinosaurLengthDeterminator!

In other words, we shouldn't even be *testing* the length! Completely delete the "huge" test. For the *other* spec test, we *do* still need to check that the isCarnivorous logic is correct, but we don't need the length stuff! Remove the second argument: `expectedIsLarge`. And then remove the asserts for it.

```
69 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 51
52 public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsCarnivorous)
53 {
54     $dinosaur = $this->factory->growFromSpecification($spec);
55
56     $this->assertSame($expectedIsCarnivorous, $dinosaur->isCarnivorous(), 'Diets do not match!');
57 }
58
... lines 59 - 67
68 }
```

In the data provider, remove the second argument from each test case.

```
69 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 58
59 public function getSpecificationTests()
60 {
61     return [
62         // specification, is large, is carnivorous
63         ['large carnivorous dinosaur', true],
64         'default response' => ['give me all the cookies!!!', false],
65         ['large herbivore', false],
66     ];
67 }
68 }
```

Try the tests again!

```
● ● ●
$ ./vendor/bin/phpunit
```

Woh! They're fixed! The tests *pass*! And this wasn't a hack: our test is now *correctly* only worried about testing the logic inside the factory class itself.

But! There's a lot more you can do with mocks - they're not *just* dummy objects. They can be trained!

Chapter 14: Mocks: Control the Return Value

We know that in `DinosaurFactoryTest`, we don't need to worry about testing the length anymore because that's done inside `DinosaurLengthDeterminator`'s test. Every test class can stay focused. Which is important when there's dinosaurs running around.

But... what if we accidentally *forgot* to call the length determinator? Like, we temporarily set the length to a hardcoded value... but forgot to fix it! Well, if you run your tests... surprise! They pass.

If the possibility of making this mistake scares you... don't worry! This is something we can test for!

[willReturn\(\)](#)

Open up `DinosaurFactoryTest`.

When you create a mock object, by default, PHPUnit overrides *all* of its methods and makes each return null... or maybe zero or an empty string, depending on the return type of the function. But, you can *teach* your mock object to return *different* values. You can say:

Hey! When somebody calls this method, don't run the real logic, but *do* return this value.

Then, we can test that *this* value *is* in fact set as the length.

To get this setup, create a new property called `lengthDeterminator`. And then set our mock onto that. This will give us access to the mock down inside the test functions.

```
78 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 19
20     private $lengthDeterminator;
... line 21
22     public function setUp()
23     {
... line 24
25         $this->factory = new DinosaurFactory($this->lengthDeterminator);
26     }
... lines 27 - 76
77 }
```

To get auto-completion, add `@var` and then `\PHPUnit_Framework_MockObject_MockObject`.

```
78 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 16
17     /**
18      * @var \PHPUnit_Framework_MockObject_MockObject
19      */
20     private $lengthDeterminator;
... lines 21 - 76
77 }
```

Now, scroll down to the specification test. *Before* we call `growFromSpecification`, we can *train* the length determinator. How?

Use `$this->lengthDeterminator->method()` and then `getLengthFromSpecification`: this is the name of the method that we call and want to control.

```
78 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 56
57     public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsCarnivorous)
58     {
59         $this->lengthDeterminator->method('getLengthFromSpecification')
... lines 60 - 65
66     }
... lines 67 - 76
77 }
```

Next, chain *off* of that with `->willReturn(20)`.

```
78 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 56
57     public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsCarnivorous)
58     {
59         $this->lengthDeterminator->method('getLengthFromSpecification')
60         ->willReturn(20);
... lines 61 - 65
66     }
... lines 67 - 78
```

That's it! *Whenever* that method is called, it will return 20. And that means, at the bottom, we can assert that 20 should match `$dinosaur->getLength()`.

```
78 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 56
57     public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsCarnivorous)
58     {
... lines 59 - 64
65         $this->assertSame(20, $dinosaur->getLength());
66     }
... lines 67 - 76
77 }
```

If it does *not*... something is fishy! Try the tests!

```
● ● ●
$ ./vendor/bin/phpunit
```

Yes! They *fail*! Go back to `DinosaurFactory`, and fix the bad length code.

```

47 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 7
8  class DinosaurFactory
9  {
... lines 10 - 21
22  public function growFromSpecification(string $specification): Dinosaur
23  {
... lines 24 - 25
26      $length = $this->lengthDeterminator->getLengthFromSpecification($specification);
... lines 27 - 35
36  }
... lines 37 - 45
46  }

```

Run the tests again! Of course *now*, they pass.

More ways to Return

This makes our test more *strict*. You might think that's *definitely* great! But... that's not *always* true! Instead of simply testing the return value of the method, we're testing *how* the code is written internally... which in some ways, we should *not* care about: that's the business of the function. All *we* are *supposed* to care about is the return value. Writing stricter tests take more time, and will break accidentally more often.

So whether or not you will choose to control the return value of a mock is up to you. Sometimes, when something is *super* important, a strict test is *awesome*. And also, pretty often, you'll *need* to control the return value to even make your method *work*.

In addition to `willReturn`, there are a few other ways to control the return value.

Google for "phpunit willreturn" and look for the "Test Doubles" documentation. Look inside this page for `willReturn()` to find a few examples. Another method is `returnValueMap()`... which is a little weird, but allows you to map different return values for different input *arguments*. That is important if you call the same method multiple times with different values.

Oh, and in this case, the code is `->will()` and then `$this->returnValueMap()`. But there's also a single method called `willReturnValueMap()`: each of these return methods can be called with both styles.

There's also one called `willReturnCallback()` where you can pass a callback and return whatever value you want. It's got the power of the value map... but is way less weird.

Ok, there's *one* more cool thing you can do with a mock: let's see it next!

Chapter 15: Mocks: expects() Assert Method is Called Correctly

Go back to DinosaurFactory. Time to create another bug! Comment out the length line again. This time, *do* call the method... but instead of the spec, just pass foo. Whoops! We're calling the right method... but we've passed the wrong argument!

What happens to the tests?

```
$ ./vendor/bin/phpunit
```

Yep... they pass! *If* this possibility frightens you more than the sound of raptor claws on the door... We can also cover this in the test.

In addition to saying that `getLengthFromSpecification()` should return 20, you can *also* say that the method must be called exactly *once* or with some *exact* arguments. Then, if the method is *not* called once or is called with different arguments, the test will *fail*.

Move the `method()` call onto the next line. Then add `->expects($this->once())`. The method must *now* be called *exactly* once.

```
80 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 56
57     public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsCarnivorous)
58     {
59         $this->lengthDeterminator->expects($this->once())
60         ->method('getLengthFromSpecification')
... lines 61 - 67
68     }
... lines 69 - 78
79 }
```

Then, after `method()`, add `->with($spec)`. This `with()` function is pretty sweet: if the real method accepts three arguments, then you'll pass those three arguments to `with()`. In this case, if the value passed to the first argument of `getLengthFromSpecification()` does *not* match `$spec`, the test will fail.

```
80 lines | tests/AppBundle/Factory/DinosaurFactoryTest.php
... lines 1 - 9
10 class DinosaurFactoryTest extends TestCase
11 {
... lines 12 - 56
57     public function testItGrowsADinosaurFromSpecification(string $spec, bool $expectedIsCarnivorous)
58     {
59         $this->lengthDeterminator->expects($this->once())
... line 60
61         ->with($spec)
62         ->willReturn(20);
... lines 63 - 67
68     }
... lines 69 - 78
79 }
```

This *should* finally kill the test. Try it!

```
$ ./vendor/bin/phpunit
```

Yes! Failure because, for `getLengthFromSpecification()`, large herbivore does not match the actual value: foo.

Awesome! Move back to `DinosaurFactory`, and re-fix the length line. Double-check that this fixes things. It does!

```
47 lines | src/AppBundle/Factory/DinosaurFactory.php
... lines 1 - 7
8  class DinosaurFactory
9  {
... lines 10 - 21
22  public function growFromSpecification(string $specification): Dinosaur
23  {
... lines 24 - 25
26      $length = $this->lengthDeterminator->getLengthFromSpecification($specification);
... lines 27 - 35
36  }
... lines 37 - 45
46 }
```

This is *really* cool stuff. But you should *not* use it everywhere. Remember: at this point, we're really testing how the internal code of the method is written. The more we do this, the more often the tests will break when really... they shouldn't. Basically, we're micro-managing our own code!

For example, what if we refactor the method and decide to call `getLengthFromSpecification()` two times? The tests will break! Sure, our code might now be a little inefficient... but as long as we *still* create the correct `Dinosaur`, shouldn't the tests pass? You need to decide how strict each test should be.

[Asserting Different Number of Method Calls](#)

And if you *don't* care how many times a method is called, you can use `$this->any()` instead. Back on the Test Doubles documentation, search for `once()`... and find the next result until you see a list of "matchers".

These are classes, but there's a shortcut to use each, like `$this->any()`, `$this->never()`, `$this->atLeastOnce()` and `$this->exactly()`.

[Smarter Argument Asserts](#)

You can also do a little bit of magic for the `with()` method. Go back to the docs and search for "anything". Cool! This is an example where the method should be called with *three* arguments. But instead of passing the exact values, we assert that the first argument is greater than zero, the second is a string that contains the word "Something" and we don't care *at all* what is passed as the third argument.

Search once more for `callback()`. This is the most *powerful* option: you create your *own* callback and do whatever logic you want to determine if the argument passed is valid.

We've got mocking down! But it's *so* important that I want to go through one more, fully-featured and classic example.

Chapter 16: Full Mock Example

Mocking is so important... and honestly... pretty fun. I think we should code through another example. In some ways... an even *better* and more common example.

Here's the setup: we're going to need a lot of dinosaurs, a lot of enclosures and even more security. Instead of creating these by hand *each* time a new batch of adorable dinosaurs arrives, let's create a service that can do it all for us.

If you downloaded the source code, then in the tutorial/ directory, you should have an EnclosureBuilderService class. Copy that and paste it into our Service directory. This has just one public function: you pass that number of security systems and the number of dinosaurs, and *it* takes care of creating those security systems, creating the dinosaurs and putting everything together inside a new Enclosure.

```
71 lines | src/AppBundle/Service/EnclosureBuilderService.php
... lines 1 - 2
3 namespace AppBundle\Service;
... lines 4 - 9
10 class EnclosureBuilderService
11 {
12     /**
13      * @var EntityManagerInterface
14      */
15     private $entityManager;
16
17     /**
18      * @var DinosaurFactory
19      */
20     private $dinosaurFactory;
21
22     public function __construct(
23         EntityManagerInterface $entityManager,
24         DinosaurFactory $dinosaurFactory
25     )
26     {
27         $this->entityManager = $entityManager;
28         $this->dinosaurFactory = $dinosaurFactory;
29     }
30
31     public function buildEnclosure(
32         int $numberOfSecuritySystems = 1,
33         int $numberOfDinosaurs = 3
34     ): Enclosure
35     {
36         $enclosure = new Enclosure();
37
38         $this->addSecuritySystems($numberOfSecuritySystems, $enclosure);
39
40         $this->addDinosaurs($numberOfDinosaurs, $enclosure);
41
42         return $enclosure;
43     }
44
45     private function addSecuritySystems(int $numberOfSecuritySystems, Enclosure $enclosure)
```

```

46     {
47         $securityNames = ['Fence', 'Electric fence', 'Guard tower'];
48         for ($i = 0; $i < $numberOfSecuritySystems; $i++) {
49             $securityName = $securityNames[array_rand($securityNames)];
50             $security = new Security($securityName, true, $enclosure);
51
52             $enclosure->addSecurity($security);
53         }
54     }
55
56     private function addDinosaurs(int $numberOfDinosaurs, Enclosure $enclosure)
57     {
58         $lengths = ['small', 'large', 'huge'];
59         $diets = ['herbivore', 'carnivorous'];
60         // We should not mix herbivore and carnivorous together,
61         // so use the same diet for every dinosaur.
62         $diet = $diets[array_rand($diets)];
63
64         $length = $lengths[array_rand($lengths)];
65         $specification = "{$length} {$diet} dinosaur";
66         $dinosaur = $this->dinosaurFactory->growFromSpecification($specification);
67
68         $enclosure->addDinosaur($dinosaur);
69     }
70 }

```

For this... we're going to cheat and *not* do TDD because... well... I *just* gave you the code. So let's add the test: in the Service directory, create a new `EnclosureBuilderServiceTest`. And inside, public function `testItBuildsAndPersistsEnclosure()`.

```

19 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php
... lines 1 - 7
8  class EnclosureBuilderServiceTest extends TestCase
9  {
10     public function testItBuildsAndPersistsEnclosure()
11     {
12         ... lines 12 - 16
17     }
18 }

```

And *this* time, let's make sure it extends `TestCase` from PHPUnit. At first, the test is pretty simple: create a new `EnclosureBuilderService()`. This has two required constructor arguments... but let's ignore those at first and finish the test. Add `$enclosure = $builder->buildEnclosure()` with, how about, 1 security system and 2 dinosaurs.

```

19 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php
... lines 1 - 7
8  class EnclosureBuilderServiceTest extends TestCase
9  {
10     public function testItBuildsAndPersistsEnclosure()
11     {
12         $builder = new EnclosureBuilderService();
13         $enclosure = $builder->buildEnclosure(1, 2);
14         ... lines 14 - 16
17     }
18 }

```

Below this, just assert that this has the right stuff: `$this->assertCount()` that 1 matches the count of `$enclosure->getSecurities()`. That method does not exist yet. And then `assertCount()` that 2 matches the count of

\$enclosure->getDinosaurs()).

```
19 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php
... lines 1 - 9
10     public function testItBuildsAndPersistsEnclosure()
11     {
... lines 12 - 13
14
15         $this->assertCount(1, $enclosure->getSecurities());
16         $this->assertCount(2, $enclosure->getDinosaurs());
17     }
... lines 18 - 19
```

We *could* test more, but this is pretty good! It tests that the core functionality works correctly. Later, if we think of some edge-case that could happen, we can add more.

Ok, find the Enclosure class, scroll to the bottom, and add the missing public function getSecurities(), which should return a Collection. Return \$this->securities.

```
84 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 14
15 class Enclosure
16 {
... lines 17 - 78
79     public function getSecurities(): Collection
80     {
81         return $this->securities;
82     }
83 }
```

Adding the Basic Mocks

Other than the missing constructor arguments, the test looks happy! But somehow, we need to pass the builder an EntityManagerInterface and a DinosaurFactory. These are both services, so they should be *mocked*, instead of created manually. That becomes even *more* obvious if you think about *trying* to create these objects. The EntityManager requires a database connection... and we definitely do *not* want to instantiate all of that. And even DinosaurFactory *itself* requires a DinosaurLengthDeterminator... so creating a new factory would take some work... *too* much work.

You can start to see why mocking makes life so much easier.

Back in the test, add \$em = \$this->createMock(). The argument expects an EntityManagerInterface. So that's what we'll use here: EntityManagerInterface::class. Yep, you can *totally* mock an interface. Then add \$dinosaurFactory = \$this->createMock(DinosaurFactory::class).

```
24 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php
... lines 1 - 9
10 class EnclosureBuilderServiceTest extends TestCase
11 {
12     public function testItBuildsAndPersistsEnclosure()
13     {
14         $em = $this->createMock(EntityManagerInterface::class);
15         $dinoFactory = $this->createMock(DinosaurFactory::class);
... lines 16 - 21
22     }
23 }
```

Pass both arguments into new EnclosureBuilderService().

24 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php

```
... lines 1 - 11
12     public function testItBuildsAndPersistsEnclosure()
13     {
    ... lines 14 - 16
17         $builder = new EnclosureBuilderService($em, $dinoFactory);
    ... lines 18 - 21
22     }
    ... lines 23 - 24
```

I'm not worried about controlling the return values: I'm trying to do as little work as possible so that the test will run and the asserts at the bottom can do their job. It *may* turn out that we *do* need to control some return values so that the function can run... but... let's find out! Run the tests:

```
$ ./vendor/bin/phpunit
```

The test *fails*! We're expecting some actual size 1 to match 2, on line 20. For some reason, we're only getting back *one* Dinosaur... even though we passed 2 as the argument.

Open EnclosureBuilder and scroll down to addDinosaur(). Ah! There's a *bug* in my code already! The \$numberOfDinosaurs argument is not used: we always add just *one* dinosaur.

That's great! The simple test caught the bug and the fix is easy.

Asserting growFromSpecification is Called Twice

But! If we want, we could also make the test a bit tougher before fixing this. In the test, add \$dinoFactory->expects(\$this->exactly(2)) and then ->method('growFromSpecification').

28 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php

```
... lines 1 - 11
12     public function testItBuildsAndPersistsEnclosure()
13     {
    ... lines 14 - 16
17         $dinoFactory->expects($this->exactly(2))
18             ->method('growFromSpecification')
19             ->with($this->isType('string'));
    ... lines 20 - 25
26     }
    ... lines 27 - 28
```

We could stop here: we don't *need* to also call ->with(). But if you *do* want to assert that the correct argument is passed, you can. Well actually, the exact argument is random. So the best we can do is use \$this->isType('string').

In practice, I think adding with() in this situation is a bit overkill. But it *always* depends.

Ok, I want to see these errors! Comment out the assert at the bottom. Any test failures from mocking will come *after* the test finishes running successfully. Try the tests!

```
$ ./vendor/bin/phpunit
```

Awesome! Translating from robot-speech, this says:

Expectation failed for method growFromSpecification(): we expected it to be called two times, but was actually called 1 time.

That's cool! Uncomment the "assert" in the test. Now, go into the service and fix my bug! Add a for loop where

`$i = 0; $i < $numberOfDinosaurs; $i++`. Move *all* that dino code inside.

```
73 lines | src/AppBundle/Service/EnclosureBuilderService.php
... lines 1 - 9
10 class EnclosureBuilderService
11 {
... lines 12 - 55
56     private function addDinosaurs(int $numberOfDinosaurs, Enclosure $enclosure)
57     {
58         $lengths = ['small', 'large', 'huge'];
59         $diets = ['herbivore', 'carnivorous'];
60         // We should not mix herbivore and carnivorous together,
61         // so use the same diet for every dinosaur.
62         $diet = $diets[array_rand($diets)];
63
64         for ($i = 0; $i < $numberOfDinosaurs; $i++) {
65             $length = $lengths[array_rand($lengths)];
66             $specification = "{ $length } { $diet } dinosaur";
67             $dinosaur = $this->dinosaurFactory->growFromSpecification($specification);
68
69             $enclosure->addDinosaur($dinosaur);
70         }
71     }
72 }
```

Move back to your terminal and, test!

```
$ ./vendor/bin/phpunit
```

We're green! But... there's something interesting going on. *And*, our test is weak in one important way... there's still a bug in EnclosureBuilderService! And our tests are missing it!

Chapter 17: Full Mock Example: the Sequel

There's something interesting going on. We're mocking the `growFromSpecification()` method... but we are *not* controlling its return value. And, the `addDinosaur()` method *requires* a `Dinosaur` object. So... how is that working? I mean, doesn't a mocked method return null by default? Shouldn't this blow up?

Back in the test, at the bottom, add `dump($enclosure->getDinosaurs()->toArray())`. Let's see what that looks like!

```
29 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php
... lines 1 - 9
10 class EnclosureBuilderServiceTest extends TestCase
11 {
12     public function testItBuildsAndPersistsEnclosure()
13     {
14         ... lines 14 - 25
26         dump($enclosure->getDinosaurs()->toArray());
27     }
28 }
```

Run the tests:

```
$ ./vendor/bin/phpunit
```

Woh! It holds 2 items... which are *mock* `Dinosaur` objects! That's really cool! Thanks to the PHP 7 return type on `growFromSpecification()`, PHPUnit is smart enough to create a *mock* `Dinosaur` and return that, instead of null.

That's not normally a detail you need to think about, but I want you to realize it's happening. We don't really need to, but if we want, we *could* add `->willReturn(new Dinosaur())`.

```
30 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php
... lines 1 - 9
10
... lines 11 - 12
13     public function testItBuildsAndPersistsEnclosure()
14     {
15         ... lines 14 - 17
18         $dinoFactory->expects($this->exactly(2))
19         {
20             ... line 19
21             ->willReturn(new Dinosaur())
22         }
23     }
24     ... lines 21 - 27
28 }
... lines 29 - 30
```

This time, the `dump()` from the test shows *real* `Dinosaur` objects. Rawr! Take the dump out of the test.

[The Bug: Unsaved Enclosure](#)

Ok, there's one more bug hiding inside `EnclosureBuilderService`. The *whole* point of the method is that we can call it and it will create the `Enclosure` *and* save it to the database. But look! That never happens! We inject the entity manager... and then... never use it! Whoops!

The *return* value of this method *is* correct... but really... we *also* care that the method did something *else*. We want to *guarantee* that `persist()` and `flush()` are called.

Back in the test, add `$em->expects($this->once())` with `->method('persist')`. We know that this should be called with an

instance of an Enclosure object. We don't know exactly *which* Enclosure object, but we can check the type with `$this->assertInstanceOf(Enclosure::class)`.

```
36 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php
... lines 1 - 13
14     public function testItBuildsAndPersistsEnclosure()
15     {
... lines 16 - 17
18         $em->expects($this->once())
19             ->method('persist')
20             ->with($this->assertInstanceOf(Enclosure::class));
... lines 21 - 33
34     }
... lines 35 - 36
```

Try the test!

```
● ● ●
$ ./vendor/bin/phpunit
```

There's the failure: persist should be called 1 time, but was called 0 times.

Back in Enclosurebuilder, add `$this->entityManager->persist($enclosure)`.

```
69 lines | src/AppBundle/Service/EnclosureBuilderService.php
... lines 1 - 9
10 class EnclosureBuilderService
11 {
... lines 12 - 30
31     public function buildEnclosure(
... lines 32 - 34
35     {
... lines 36 - 41
42         $this->entityManager->persist($enclosure);
... lines 43 - 44
45     }
... lines 46 - 67
68 }
```

Of course, the flush() call is still missing. In the test, check for that: `$em->expects($this->atLeastOnce())->method('flush')`.

```
39 lines | tests/AppBundle/Service/EnclosureBuilderServiceTest.php
... lines 1 - 11
12 class EnclosureBuilderServiceTest extends TestCase
13 {
14     public function testItBuildsAndPersistsEnclosure()
15     {
... lines 16 - 21
22         $em->expects($this->atLeastOnce())
23             ->method('flush');
... lines 24 - 36
37     }
38 }
```

You could also use `$this->once()`... calling flush() multiple times isn't a problem... but it *is* a bit wasteful. Make sure the test fails before we fix it:

```
$ ./vendor/bin/phpunit
```

It *does*. In the builder, add `$this->entityManager->flush()` and then... run the tests. They pass!

```
71 lines | src/AppBundle/Service/EnclosureBuilderService.php
```

```
... lines 1 - 9
```

```
10 class EnclosureBuilderService
```

```
11 {
```

```
... lines 12 - 30
```

```
31     public function buildEnclosure()
```

```
... lines 32 - 34
```

```
35     {
```

```
... lines 36 - 43
```

```
44         $this->entityManager->flush();
```

```
... lines 45 - 46
```

```
47     }
```

```
... lines 48 - 69
```

```
70 }
```

Thanks to mocking, we just created a *killer* test. Just remember: if the object you need is a service, mock it. If it's a simple model object, that's overkill: just create the object normally.

Chapter 18: Mocking with Prophecy

PHPUnit has a mocking system. But it's not the *only* mocking library available. There are two other popular ones: Mockery & Prophecy. They all do the same thing, but each has its own *feel*.

I really like Prophecy, *and* it comes with PHPUnit automatically! So let's redo the EnclosureBuilderTest with Prophecy to see how it feels.

Create a new class called EnclosureBuilderServiceProphecyTest. It will extend the normal TestCase and we can give it the same method: testItBuildsAndPersistsEnclosure().

```
20 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php
... lines 1 - 12
13 class EnclosureBuilderServiceProphecyTest extends TestCase
14 {
15     public function testItBuildsAndPersistsEnclosure()
16     {
17         ... line 17
18     }
19 }
```

Mocking Prophecy Style

Let's translate the PHPUnit mock code into Prophecy line-by-line. To create the EntityManager mock, use `$this->prophesize(EntityManagerInterface::class)`. That's pretty similar.

```
20 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php
... lines 1 - 12
13 class EnclosureBuilderServiceProphecyTest extends TestCase
14 {
15     public function testItBuildsAndPersistsEnclosure()
16     {
17         $em = $this->prophesize(EntityManagerInterface::class);
18     }
19 }
```

Next, we need to assert that `persist()` will be called once() and that it is passed an Enclosure object. *This* is where things get different... and pretty fun... Instead of thinking of `$em` as a mock, pretend it's the *real* object. Call `$em->persist()`. To make sure this is passed some Enclosure object, pass `Argument::type(Enclosure::class)`.

```
23 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php
... lines 1 - 14
15     public function testItBuildsAndPersistsEnclosure()
16     {
17         ... lines 17 - 18
19         $em->persist(Argument::type(Enclosure::class))
20         ... line 20
21     }
22     ... lines 22 - 23
```

We'll talk more about how these arguments work in a minute. Then, because we want this to be called exactly once, add `shouldBeCalledTimes(1)`.

23 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php

... lines 1 - 14

```
15     public function testItBuildsAndPersistsEnclosure()  
16     {  
    ... lines 17 - 18  
19     $em->persist(Argument::type(Enclosure::class))  
20     ->shouldBeCalledTimes(1);  
21     }
```

... lines 22 - 23

Oh, and notice that I am *not* getting auto-completion. That's because Prophecy is a super magic library, so PhpStorm doesn't really know what's going on. But actually, there are two *amazing* PhpStorm plugins that - together - *will* give you auto-completion for Prophecy... and many other things. They're called "PHP Toolbox" and "PHPUnit Enhancement". I learned about these so recently, that I didn't have them installed yet for this tutorial. Thanks for the tip Stof!

Next, we need to make sure `flush()` is called at least once. That's easy: `$em->flush()->shouldBeCalled()`.

25 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php

... lines 1 - 14

```
15     public function testItBuildsAndPersistsEnclosure()  
16     {  
    ... lines 17 - 21  
22     $em->flush()->shouldBeCalled();  
23     }
```

... lines 24 - 25

Don't you love it? In addition to `shouldBeCalledTimes()` and `shouldBeCalled()`, there is also `shouldNotBeCalled()` and simply `should()`, which accepts a callback so you can do custom logic.

Mocking the DinosaurFactory

Let's keep moving: add the `DinosaurFactory` with `$dinoFactory = $this->prophesize()` and `DinosaurFactory::class`.

27 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php

... lines 1 - 14

```
15     public function testItBuildsAndPersistsEnclosure()  
16     {  
    ... lines 17 - 23  
24     $dinoFactory = $this->prophesize(DinosaurFactory::class);  
25     }
```

... lines 26 - 27

Here, we need to make sure that the `growFromSpecification` method is called exactly two times with a string argument and returns a dinosaur. Ok! Start with `$dinoFactory->growFromSpecification()`.

Here's how the arguments part *really* works. If you don't care what arguments are passed to the method, just leave this blank. But if you *do* care, then you need to pass *all* of the arguments here, as if you were *calling* this method.

For example, imagine the method accepts *three* arguments. If we passed `foo`, `bar`, `baz` here, this would make sure that the method was called with exactly these three args.

Our situation is a bit trickier: we don't know the *exact* argument, we only know that it should be a string. To check that, use `Argument::type('string')`.

32 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php

... lines 1 - 14

```
15     public function testItBuildsAndPersistsEnclosure()
16     {
    ... lines 17 - 25
26     $dinoFactory
27     ->growFromSpecification(Argument::type('string'))
    ... lines 28 - 29
30     }
```

... lines 31 - 32

There are a few other useful methods on this Argument class. The most important is Argument::any(). You'll need this if you want to assert that *some* of your arguments match a value, but you don't care what value is passed for the *other* arguments.

The most powerful is that(), which accepts an all-powerful callback as an argument.

Next, this method should be called 2 times. No problem: ->shouldBeCalledTimes(2). And finally, it should return a new Dinosaur object. And that's the same as in PHPUnit: ->willReturn(new Dinosaur()). The other 2 useful functions are willThrow() to make the method throw an exception and will(), which accepts a callback so you can completely control the return value.

32 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php

... lines 1 - 14

```
15     public function testItBuildsAndPersistsEnclosure()
16     {
    ... lines 17 - 25
26     $dinoFactory
27     ->growFromSpecification(Argument::type('string'))
28     ->shouldBeCalledTimes(2)
29     ->willReturn(new Dinosaur());
30     }
```

... lines 31 - 32

And... ye! That's it! I'll copy the rest of the test and paste it. Re-type the e on EnclosureBuilderService to add the use statement on top.

41 lines | tests/AppBundle/Service/EnclosureBuilderServiceProphecyTest.php

... lines 1 - 7

```
8     use AppBundle\Service\EnclosureBuilderService;
    ... lines 9 - 12
13     class EnclosureBuilderServiceProphecyTest extends TestCase
14     {
15         public function testItBuildsAndPersistsEnclosure()
16         {
    ... lines 17 - 30
31         $builder = new EnclosureBuilderService(
32             $em->reveal(),
33             $dinoFactory->reveal()
34         );
35         $enclosure = $builder->buildEnclosure(1, 2);
36
37         $this->assertCount(1, $enclosure->getSecurities());
38         $this->assertCount(2, $enclosure->getDinosaurs());
39     }
40 }
```

[Revealing the Prophecy](#)

There's *one* other tiny difference in Prophecy. First, I'll break this onto multiple lines so it looks nicer. When you finally pass in your mock, you need to call `->reveal()`. On a technical level, this kind of turns your "mock builder" object into a true mock object. On a philosophical Prophecy level, this *reveals the prophecy* that the prophet prophesized.

Fun, right? If that made no sense - it's ok! The Prophecy documentation - while being a little strange - is really fun and talks a lot more about dummies, stubs, prophets and other things. If you're curious, it's worth a read.

Ok, that should be it! Find your terminal and run the test:

A terminal window with a dark background and three light gray window control buttons (minimize, maximize, close) in the top-left corner. The terminal shows a prompt character '\$' followed by the command './vendor/bin/phpunit'.

They pass! Right on the first try. So that's Prophecy: it's a bit more fun than PHPUnit and is also quite popular. If you like it better, use it!

Next, there are *many* options you can pass to the phpunit command. Let's learn about the most important ones... so that we can ignore the rest.

Chapter 19: The Important CLI Options & phpunit.xml.dist

Find your terminal. But *this* time, run phpunit with a -h option:

```
$ ./vendor/bin/phpunit -h
```

Woh! That is a *lot* of options. Ok, let's talk about *every* single one of them. Just kidding! I've never even used most of these options! But there are a few *awesome* flags that will be *very* useful.

Google for "PHPUnit cli" to find their page about this. It's a bit prettier than reading the terminal. At the top of the output, it says that you can pass one or more files or directories. That's useful to run just *one* test class or all the classes in one directory. For example, we could run just the DinosaurFactoryTest:

```
$ ./vendor/bin/phpunit tests/AppBundle/Factory/DinosaurFactoryTest.php
```

[The Symfony PHPUnit Bridge](#)

Oh, and before I forget, Google for "Symfony PHPUnit Bridge" to find a special component that lives in Symfony.com: [symfony/phpunit-bridge](#).

This is a wrapper around PHPUnit that adds a couple of extra features like deprecation reporting that will tell you about deprecated code paths that you're using during your tests.

Tip

In Symfony 4, this is the officially-recommended way to use PHPUnit. You should install this instead of installing PHPUnit directly.

Basically, after you install this, you'll use vendor/bin/simple-phpunit to activate it. It supports all the same options.

[The --filter Option](#)

Back on the PHPUnit docs, my *favorite* option by far is --filter. This let's you run just *one* test method, and it's *critical* when you're trying to debug *one* test.

If you scroll down, the docs show a bunch of examples. Usually, I copy the method name I'm testing, pass --filter and then paste that name. But you can get a lot fancier.

Let's try a few! First, re-run the test with another flag: --debug:

```
$ ./vendor/bin/phpunit tests/AppBundle/Factory/DinosaurFactoryTest.php --debug
```

The output now tells us *which* tests are running. Let's run *just* one of them:

```
$ ./vendor/bin/phpunit --filter testItGrowsADinosaurFromASpecification --debug
```

And... yes! It ran *three* tests, because this one method has a data provider. But sometimes, you'll need to debug *just* one test case of a provider. Surround the method name in quotes, and then add #1:

```
$ ./vendor/bin/phpunit --filter 'testItGrowsADinosaurFromASpecification #1' --debug
```

Ah, so cool! You can also use `@default` response - that's the test case that we gave a special name.

```
$ ./vendor/bin/phpunit --filter 'testItGrowsADinosaurFromASpecification @default response' --debug
```

For the *longest* time, I didn't know you could do this. I *love* it.

[Stopping on Failure or Error](#)

If you're running a *lot* of tests, you can tell PHPUnit to stop *immediately* when one of them has an error or fails... instead of waiting for *all* of them to finish running.

To do that, use the `--stop-on-failure` and `--stop-on-error` options:

```
$ ./vendor/bin/phpunit --stop-on-failure --stop-on-error
```

We don't have any errors - yes! - but you get the idea!

[The phpunit.xml.dist File](#)

There are *many* other options... but you can do even *more* in the configuration file! And we already have one! PHPUnit automatically looks for a `phpunit.xml` file and then a `phpunit.xml.dist` file.

This configures a few *really* important things... like bootstrap. Thanks to this, PHPUnit requires the autoloader before running the tests. But there is so much more you can do: tweak `php.ini` settings, set environment variables, configure test suites or tweak code coverage setup.

Most of the time, you'll have just one test suite... which means that all the test are always executed. But you could, for example, separate your unit, integration and functional tests into different suites so that you could run them independently. That's kind of cool, because integration and functional tests are much slower than unit tests.

Let's do one quick example: add a test suite called `entities` and set its directory to `tests/*Bundle/Entity`.

To run this suite, use:

```
$ ./vendor/bin/phpunit --testsuite entities --debug
```

It runs *only* those tests.

Ok, enough of this! Let's get back to work and talk about integration tests: what they are, and when they can save your life.

Chapter 20: Integration Tests

Isn't *mocking* awesome? Yes! Except... when it's not. In a unit test, we use mocking so that each class can be tested in complete isolation: no database, no API calls and no friendly dinner parties. But sometimes... if you mock everything... there's nothing *really* left to test

For example, how would you test that a complex query in a Doctrine repository works? If you *mock* the database connection then... I guess you could test that the query string you wrote looks ok? That's silly! The *only* way to *truly* test this method is to run that query against a *real* database.

Here's the deal: sometimes, when you think about testing a class, you start to realize that if you mock all the dependencies... then the test becomes worthless! In these cases, you need an *integration* test.

Setting up the Database

Let's jump in! EnclosureBuilderService already has a unit test. But since it talks to the database, if we *really* want to make sure it works, we need a test where it... *actually* talks to the database!

First, we need to finish our entities. Find Security and copy the id field. Open Dinosaur and paste this in. Do the same for Enclosure. We haven't needed these yet because we haven't touched the database at all.

```
82 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 15
16 /**
17  * @ORM\Id
18  * @ORM\GeneratedValue(strategy="AUTO")
19  * @ORM\Column(type="integer")
20  */
21 private $id;
... lines 22 - 80
81 }
```

```
90 lines | src/AppBundle/Entity/Enclosure.php
... lines 1 - 14
15 class Enclosure
16 {
17 /**
18  * @ORM\Id
19  * @ORM\GeneratedValue(strategy="AUTO")
20  * @ORM\Column(type="integer")
21  */
22 private $id;
... lines 23 - 88
89 }
```

Now, go to your terminal and create the database:

```
$ php bin/console doctrine:database:create
```

Huh, I already have one. Lucky me! Create the schema:

```
$ php bin/console doctrine:schema:create
```

Hello Integration Test

Now to the integration test! Create a new class: `EnclosureBuilderServiceIntegrationTest`. I don't always create a separate class for integration tests: it's up to you. Unit tests and integration tests can actually live next to each other in the same test class. Unlike herbivore and carnivore dinosaurs who should really each get their own enclosure.

This time, instead of `TestCase`, extend `KernelTestCase`.

```
18 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 2
3 namespace Tests\AppBundle\Service;
... lines 4 - 7
8 class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
... lines 9 - 18
```

This is not *that* crazy: `KernelTestCase` itself extends `TestCase`: so we have all the normal methods. But it *also* has a few new methods to help us boot Symfony's container. And *that* will give us access to our *real* services.

Add the test method: public function `testItBuildsEnclosureWithDefaultSpecifications()`:

```
18 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 7
8 class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
9 {
10     public function testItBuildsEnclosureWithDefaultSpecifications()
11     {
... lines 12 - 15
16     }
17 }
```

Hmm, that's a big name!

Booting & Fetching the Container

Here is the key difference between a unit test and an integration test: instead of creating the `EnclosureBuilderService` and passing in mock dependencies, we'll boot Symfony's container and ask *it* for the `EnclosureBuilderService`. And of course, *that* will be configured to talk to our real database. This makes integration tests less "pure" than unit tests: if an integration test fails, the problem could live in *multiple* different places - not just in *this* class. And also, integration tests are *way* slower than unit tests. Together, this makes them less hipster than unit tests. Despite my love for being hipster, I'll concede that integration tests are *really* helpful.

To use the real services, first call `self::bootKernel()` to... um... boot Symfony's "kernel": its "core". Now we can say `$enclosureBuilderService = self::$kernel->getContainer()->get()` and the service's id: `EnclosureBuilderService::class`.

```

18 lines | tests/AppBundle\Service\EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 7
8  class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
9  {
10     public function testItBuildsEnclosureWithDefaultSpecifications()
11     {
12         self::bootKernel();
13
14         $enclosureBuilderService = self::$kernel->getContainer()
15             ->get(EnclosureBuilderService::class);
16     }
17 }

```

But before we do *anything* else... there's a surprise! Find your terminal and run phpunit with `--filter`. Copy the method's name and paste it:

```
$ ./vendor/bin/phpunit --filter testItBuildsEnclosureWithDefaultSpecifications
```

Fetching Private Services

Woh! It *explodes*!

You have requested a non-existent service `AppBundle\Service\EnclosureBuilderService`.

That's weird because... in `app/config/services.yml`, we're using the service auto-registration code from Symfony 3.3, which registers each class as a service... and uses the *class* name as the service id. So why does it say the service isn't found?

Because... all services are *private* thanks to the `public: false`. This is actually very important to how Symfony works - you can learn more about it in our Symfony 3.3 tutorial. But the point is, when a service is `public: false`, it means that you cannot fetch it directly from the container. Normally, that's no problem! We use dependency injection everywhere. Well... everywhere *except* our tests.

How do we fix this? Open `app/config/config_test.yml`. In Symfony 4, you should open or create `config/services_test.yml`. Add the `services` key and use `_defaults` below with `public: true`.

```

23 lines | app/config/config_test.yml
... lines 1 - 3
4  services:
5    _defaults:
6      public: true
... lines 7 - 23

```

Then, we're going to create a service *alias*. Back in the test, copy the entire class name - which is the service id. Over in `config_test.yml`, add `test.` and then paste. Set this to `@` and paste again.

```

23 lines | app/config/config_test.yml
... lines 1 - 3
4  services:
... lines 5 - 7
8    test.AppBundle\Service\EnclosureBuilderService: '@AppBundle\Service\EnclosureBuilderService'
... lines 9 - 23

```

This creates a public *alias*: even though the original service is private, we can use this new test. service id to fetch our original service out of the container.

Try it! Back in the test, inside `get()`, add `test.` and *then* the class name.

```

18 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 7
8  class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
9  {
10     public function testItBuildsEnclosureWithDefaultSpecifications()
11     {
... lines 12 - 13
14         $enclosureBuilderService = self::$kernel->getContainer()
15             ->get('test.'.EnclosureBuilderService::class);
16     }
17 }

```

Move over and try the test again!

```

$ ./vendor/bin/phpunit --filter testItBuildsEnclosureWithDefaultSpecifications

```

Ha! It works! It shows "Risky" because we don't have any assertions. But it did *not* blow up.

Adding the Database Assertions

Let's finish the thing! Above the variable, I'll add some inline documentation so that PhpStorm gives me auto-completion. Now, call the `->buildEnclosure()` method. We'll use the default arguments. That should create 1 Security and 3 Dinosaur entities.

```

44 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 10
11 class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
12 {
13     public function testItBuildsEnclosureWithDefaultSpecifications()
14     {
... lines 15 - 19
20         $enclosureBuilderService->buildEnclosure();
... lines 21 - 41
42     }
43 }

```

And... yea! All we need to do now is count the results in the database to make sure they're correct! First, fetch the EntityManager with `self::$kernel->getContainer()` then `->get('doctrine')->getManager()`. I'll also add inline phpdoc above this to help code completion.

```

44 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 12
13     public function testItBuildsEnclosureWithDefaultSpecifications()
14     {
... lines 15 - 21
22         /** @var EntityManager $em */
23         $em = self::$kernel->getContainer()
24             ->get('doctrine')
25             ->getManager();
... lines 26 - 41
42     }
... lines 43 - 44

```

To count the results, I'll paste in some code: this accesses the Security repository, counts the results and calls `getSingleScalarResult()` to return *just* that number. After this, use `$this->assertSame()` to assert that 1 will match `$count`. If they don't match, then the "Amount of security systems is not the same". And you should look over your shoulder for escaped

raptors!

```
44 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 12
13     public function testItBuildsEnclosureWithDefaultSpecifications()
14     {
... lines 15 - 26
27         $count = (int) $em->getRepository(Security::class)
28             ->createQueryBuilder('s')
29             ->select('COUNT(s.id)')
30             ->getQuery()
31             ->getSingleScalarResult();
32
33         $this->assertSame(1, $count, 'Amount of security systems is not the same');
... lines 34 - 41
42     }
... lines 43 - 44
```

Copy all of that and repeat for Dinosaur. Change the class name, and I'll change the alias to be consistent. Update the message to say "dinosaurs" and this time - thanks to the default arguments in buildEnclosure() - there should be 3.

```
44 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 12
13     public function testItBuildsEnclosureWithDefaultSpecifications()
14     {
... lines 15 - 34
35         $count = (int) $em->getRepository(Dinosaur::class)
36             ->createQueryBuilder('d')
37             ->select('COUNT(d.id)')
38             ->getQuery()
39             ->getSingleScalarResult();
40
41         $this->assertSame(3, $count, 'Amount of dinosaurs is not the same');
42     }
... lines 43 - 44
```

Ok team! We're done! Try the test!

```
$ ./vendor/bin/phpunit --filter testItBuildsEnclosureWithDefaultSpecifications
```

It works! We're geniuses! Nothing could ever go wrong! Run the test again:

```
$ ./vendor/bin/phpunit --filter testItBuildsEnclosureWithDefaultSpecifications
```

It fails! Suddenly there are 2 security systems in the database! And each time you execute the test, we have more: 3, 4, 5! It's easy to see what's going on: each test adds more and more stuff to the database.

As soon as you talk to the database, you have a new responsibility: you need to control *exactly* how the database looks.

Let's talk about that next.

Chapter 21: Clearing the Database

Each time we run the test, it adds more and more entries to the database. Our test is *not* reliable: it *depends* on what the database looks like when it starts. The solution is simple: we *must* control the data in the database at the start of every test.

Creating a Test Database

Step one to accomplishing this is to use a *different* database for our test environment. Actually, this is mostly for convenience: using the same database for testing and development is annoying. One minute, you're coding through something awesome in your browser, then you run your tests, then back in the browser, all your nice data is gone or totally different! Isolation in this instance is nice.

To do that, go back to `config_test.yml`. In a Symfony 4 Flex application, you should create a `config/packages/test/doctrine.yaml` file since this will contain Doctrine configuration that you *only* want to use in the test environment.

Inside, anywhere, add doctrine, dbal then url set to `sqlite:///kernel.project_dir%/var/data/test.sqlite`.

```
28 lines | app/config/config_test.yml
... lines 1 - 23
24 doctrine:
25   dbal:
26     url: 'sqlite:///kernel.project_dir%/var/data/test.sqlite'
... lines 27 - 28
```

This will override the settings in `app/config/config.yml` - the stuff under doctrine. With the new config, we're completely replacing this stuff and saying "Hey! Use an sqlite, flat-file database!".

Why Sqlite? It's simple to setup and you can use some tricks to speed up your tests. We'll see that in a few minutes.

Oh, and make sure you have the % sign at the *end* of `kernel.project_dir`!

Now, find your terminal and create the `var/data` directory:

```
$ mkdir var/data
```

Next, create the schema

```
$ php bin/console doctrine:schema:create --env=test
```

And, congrats! You are the owner of a fancy new `var/data/test.sqlite` file! Take good care of it.

Clearing the Database before Tests

At this point, not much has changed really. Our tests will still pass *one* time, but will fail each time after. We haven't *actually* fixed the problem yet!

How can we? The best way is to *fully* empty the database at the beginning of each test. This would *certainly* put our database into a known state: empty! Then, if we *do* need any data before running the test, we can manually add it *in* the test. It's not super fancy, but it keeps everything really clear.

Like most good things in life, there are two ways to do this. First, if you downloaded the course code, then in the `tutorial/tests` directory, you'll find an `EnclosureBuilderServiceIntegrationTest.php` file. Copy the `truncateEntities()` method and paste that into your test class.


```

83 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 11
12 class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
13 {
... lines 14 - 50
51 private function truncateEntities(array $entities)
52 {
53     $connection = $this->getEntityManager()->getConnection();
54     $databasePlatform = $connection->getDatabasePlatform();
55
56     if ($databasePlatform->supportsForeignKeyConstraints()) {
57         $connection->query('SET FOREIGN_KEY_CHECKS=0');
58     }
59
60     foreach ($entities as $entity) {
61         $query = $databasePlatform->getTruncateTableSQL(
62             $this->getEntityManager()->getClassMetadata($entity)->getTableName()
63         );
64
65         $connection->executeUpdate($query);
66     }
67
68     if ($databasePlatform->supportsForeignKeyConstraints()) {
69         $connection->query('SET FOREIGN_KEY_CHECKS=1');
70     }
71 }
... lines 72 - 81
82 }

```

This is simple: pass the method an array of entities and it will empty them.

You might want to call this at the top of every test method. But another great option is to override `setUp()` and add it there. Let's empty all three entities: Enclosure, Security and Dinosaur.

```

83 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 11
12 class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
13 {
14     public function setUp()
15     {
16         self::bootKernel();
17
18         $this->truncateEntities([
19             Enclosure::class,
20             Security::class,
21             Dinosaur::class,
22         ]);
23     }
... lines 24 - 81
82 }

```

For this method to work, we need a `getEntityManager()` method. At the bottom, add private function `getEntityManager()`. Then, copy our logic from above, paste it here, and add `return`. And since you know I /love auto-completion, I'll add a `@return EntityManager`.

83 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php

... lines 1 - 72

```
73     /**
74      * @return EntityManager
75      */
76     private function getEntityManager()
77     {
78         return self::$kernel->getContainer()
79             ->get('doctrine')
80             ->getManager();
81     }
```

... lines 82 - 83

This makes truncateEntities() happy! And we can even use getEntityManager() above.

83 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php

... lines 1 - 24

```
25     public function testItBuildsEnclosureWithDefaultSpecifications()
26     {
27         ... lines 27 - 31
32         $em = $this->getEntityManager();
33         ... lines 33 - 48
49     }
```

... lines 50 - 83

Oh, and it's *really* important that we call self::bootKernel() *before* we try to access any services. The best thing to do is remove it from the test method and add it to setUp().

83 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php

... lines 1 - 13

```
14     public function setUp()
15     {
16         self::bootKernel();
17         ... lines 17 - 22
23     }
```

... lines 24 - 83

Done! Try the tests:

```
$ ./vendor/bin/phpunit --filter testItBuildsEnclosureWithTheDefaultSpecification
```

We got it! We can run them over, and over, and over again. Always green!

Using Data Fixtures

This was the more manual way to clear the database, and it gives you a bit more control. Another option is to use Doctrine's DataFixtures library.

First, install it:

```
$ composer require doctrine/data-fixtures --dev
```

When this finishes, we can delete all the logic in truncateEntities()... because *now* we have a fancy "purger" object: \$purger = new ORMPurger() and pass in the entity manager.

Then... \$purger->purge(). And yea... that's it! We can remove the \$entities argument and stop passing in the array.

64 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php

... lines 1 - 47

```
48     private function truncateEntities()
49     {
50         $purger = new ORMPurger($this->getEntityManager());
51         $purger->purge();
52     }
```

... lines 53 - 64

This loops over all of your entity objects and deletes them one by one. It will even delete them in the correct order to avoid foreign key problems. But, if you have two entities that *both* have foreign keys pointing at each other, you may still have problems.

But, this works! The tests still pass.

Before we move on, I want to show you one cool, weird trick with integration tests: I call it "partial" mocking.

Chapter 22: Partial Mocking

There's *one* more cool thing I want to show you with integration tests. It's kind of a weird, unnatural mixture of unit tests and integration tests. I love it! I call it partial mocking.

Right now, we're fetching the `EnclosureBuilderService` from the container. But there's another option. Instead, create it manually - `new EnclosureBuilderService()` - and pass in the dependencies manually. For example, pass `$this->getEntityManager()` as the first argument.

```
75 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 13
14 class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
15 {
... lines 16 - 22
23 public function testItBuildsEnclosureWithDefaultSpecifications()
24 {
... lines 25 - 32
33     $enclosureBuilderService = new EnclosureBuilderService(
34         $this->getEntityManager(),
... line 35
36     );
... lines 37 - 56
57 }
... lines 58 - 73
74 }
```

Why would we do this? It seems like more work! Because... if it's useful, we could *mock* certain dependencies. Yep! Instead of fetching the `DinosaurFactory` from the container, mock it: `$dinoFactory = $this->createMock(DinosaurFactory::class)`.

```
75 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 13
14 class EnclosureBuilderServiceIntegrationTest extends KernelTestCase
15 {
... lines 16 - 22
23 public function testItBuildsEnclosureWithDefaultSpecifications()
24 {
... lines 25 - 27
28     $dinoFactory = $this->createMock(DinosaurFactory::class);
... lines 29 - 56
57 }
... lines 58 - 73
74 }
```

Then, `$dinoFactory->expects($this->any())` - we don't really care - with `->method('growFromSpecification')` and `->willReturn(new Dinosaur())`.

```

75 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 22
23     public function testItBuildsEnclosureWithDefaultSpecifications()
24     {
... lines 25 - 28
29         $dinoFactory->expects($this->any())
30             ->method('growFromSpecification')
31             ->willReturn(new Dinosaur());
... lines 32 - 56
57     }
... lines 58 - 75

```

Pass *this* as the second argument. This isn't *particularly* useful in this situation. But sometimes, being able to mock - and *control* - one or two dependencies in an integration test is *really* awesome!

```

75 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 22
23     public function testItBuildsEnclosureWithDefaultSpecifications()
24     {
... lines 25 - 32
33         $enclosureBuilderService = new EnclosureBuilderService(
34             $this->getEntityManager(),
35             $dinoFactory
36         );
... lines 37 - 56
57     }
... lines 58 - 75

```

Ok, try the test!

```

$ ./vendor/bin/phpunit --filter testItBuildsEnclosureWithTheDefaultSpecification

```

Oh man! It fails! Weird! There is only 1 Dinosaur... but there should be 3! What's going on? This is subtle: PHPUnit is smart enough to take this *one* dinosaur object and return it each time `growFromSpecification()` is called. But to Doctrine, it looks like we're asking it to save the same *one* Dinosaur object: not three *separate* dinosaurs. The result would be a less than thrilling theme park.

The fix is to change this to `willReturnCallback()` and pass it a function. This will be called each time `growFromSpecification()` is called. And since it is passed a `$specification` argument, the callback also receives this. It's the best way to return different values based on the arguments.

```

77 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 22
23     public function testItBuildsEnclosureWithDefaultSpecifications()
24     {
... lines 25 - 28
29         $dinoFactory->expects($this->any())
... line 30
31         ->willReturnCallback(function($spec) {
... line 32
33             });
... lines 34 - 58
59     }
... lines 60 - 77

```

We don't need that in this case. We'll just say `return new Dinosaur()`. And that's it! Try the tests again.

```
77 lines | tests/AppBundle/Service/EnclosureBuilderServiceIntegrationTest.php
... lines 1 - 22
23     public function testItBuildsEnclosureWithDefaultSpecifications()
24     {
... lines 25 - 28
29         $dinoFactory->expects($this->any())
... line 30
31         ->willReturnCallback(function($spec) {
32             return new Dinosaur();
33         });
... lines 34 - 58
59     }
... lines 60 - 77
```

```
$ ./vendor/bin/phpunit --filter testItBuildsEnclosureWithTheDefaultSpecification
```

We got it! So integration tests are one of my favorite, favorite, *favorite* things because they're *very* pragmatic. When I think about testing a class, here's the logic I follow:

1. Does this class scare me? If the logic is simple, I don't test it. I *will* still make sure the new feature works through manual testing. And often, that's enough.
2. Can I unit test this class? If I mock all of the dependencies, is the scary logic still testable? For DinosaurFactory, the answer was yes: we mocked the DinosaurLengthDeterminator, but we were still able to test that the specification string is parsed correctly to create carnivorous and non-carnivorous dinosaurs.
3. If the class cannot be unit tested, then I write an integration test. This is pretty common in my apps: each individual "unit" is often pretty simple. But when you integrate them all together, that's scary enough to need a test.

Ok, on to functional testing!

Chapter 23: Functional Tests

The last type of test is called a *functional* test. And it's *way* different than what we've seen so far. With unit and integration tests, we call methods on our code and test the output. But with a functional test, you command a browser, which surfs to your site, clicks on links, fills out forms and asserts things it sees on the page. Yep, you're testing the interface that your users *actually* use.

Oh, and functional tests also apply if you're building an API. It's the same idea: you would use an HTTP client to make real HTTP requests to your API and assert the output.

[LiipFunctionalTestBundle](#)

First, to give us some magic, I want to install a special bundle: Google for LiipFunctionalTestBundle.

This bundle is *not* needed to write functional tests. But, it has a collection of optional, extra goodies!

Copy the composer require line, move over to your terminal, and paste:

```
$ composer require --dev liip/functional-test-bundle
```

Tip

If you are using PHPUnit 7+ or Symfony 4, you need to require version 2.0 of this bundle. At this time, 2.0 is still alpha, and needs to be installed specifically with: `composer require --dev liip/functional-test-bundle:~2.0@alpha`

If you're using Symfony 3, make sure you've installed PHPUnit 6.3 and install version 1 of this bundle so that everything can play together nicely: `composer require --dev liip/functional-test-bundle:^1.9`

[Functional Test Setup](#)

Functional tests *look* like unit tests at first: they use PHPUnit in the exact way we've been seeing. But instead of writing one test class per PHP class, you'll usually create one test class per *controller* class.

It doesn't have much yet, but we're going to functionally test our homepage. Since the code behind this lives in `DefaultController`, let's create a `Controller` directory in `tests` and add a new `DefaultControllerTest` class.

But now, instead of extending `TestCase` or `KernelTestCase`, extend `WebTestCase`. But wait! There are two! The normal base class is the one from `FrameworkBundle`. *It* actually extends `KernelTestCase`, which means we have all the same tools as integration tests. But, it adds a few methods to help create a *client* object: a special object we'll use to make requests into our app.

Today we'll choose `WebTestCase` from `LiipFunctionalTestBundle`. No surprise, this class *itself* extends the normal `WebTestCase`. Then, it adds a bunch of optional magic.

```
18 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 4
5  use Liip\FunctionalTestBundle\Test\WebTestCase;
6
7  class DefaultControllerTest extends WebTestCase
8  {
... lines 9 - 16
17 }
```

[TDD & The Functional Test](#)

Let's add the first test: `public function testEnclosuresAreShownOnTheHomepage()`. Right now, the homepage is empty. But

in a minute, we're going to render all of the enclosures. So let's do a little TDD testing! Start by creating a client with `$client = $this->makeClient()`. This method comes from `LiipFunctionalTestBundle`, but is just a wrapper around Symfony's normal `static::createClient()`. The version from the bundle just adds some optional authentication magic.

```
18 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 6
7  class DefaultControllerTest extends WebTestCase
8  {
9      public function testEnclosuresAreShownOnHomepage()
... line 10
11     $client = $this->makeClient();
... lines 12 - 15
16 }
17 }
```

Next, make a request! `$crawler = $client->request('GET', '/')` to go to the homepage. We'll talk more about this Crawler object in a few minutes. Then, the simplest test is to say `$this->assertStatusCode(200)` and pass `$client`. But even this is just a shortcut to make sure 200 matches `$client->getResponse()->getStatusCode()`.

```
18 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 8
9  public function testEnclosuresAreShownOnHomepage()
... lines 10 - 12
13     $crawler = $client->request('GET', '/');
14
15     $this->assertStatusCode(200, $client);
16 }
... lines 17 - 18
```

And yea... the first part of the test is done! This *at least* makes sure our page isn't broken!

[Finishing Installing LiipFunctionalTestBundle](#)

But before we try it, we need to finish installing the bundle. Copy the 3 bundle lines, open `AppKernel` and paste them there.

```
60 lines | app/AppKernel.php
... lines 1 - 6
7  class AppKernel extends Kernel
8  {
9      public function registerBundles()
10     {
... lines 11 - 21
22     if (in_array($this->getEnvironment(), ['dev', 'test'], true)) {
... lines 23 - 31
32         if ('test' === $this->getEnvironment()) {
33             $bundles[] = new LiipFunctionalTestBundle();
34         }
35     }
... lines 36 - 37
38 }
... lines 39 - 58
59 }
```

We also need to add one line to `config_test.yml`.


```
29 lines | app/config/config_test.yml
```

```
... lines 1 - 27
```

```
28 liip_functional_test: ~
```

If you're using Symfony Flex, these steps should eventually be done for you. I say eventually, because - at this moment - Symfony 4 support is still being added to the bundle.

Ok! Let's try it! Find your terminal, run phpunit, and point it at the new controller:

```
$ ./vendor/bin/phpunit tests/AppBundle/Controller/DefaultControllerTest.php
```

Yes! It passes!

Symfony's Client Versus Mink Versus Others

But... what just happened exactly? Did our code just make a *real* HTTP request to our app... just like if we refreshed it in the browser? Well... not *quite*.

Remember: in a functional test, we use PHP to command a browser and tell it to go to a page, click on links and fill out forms. But, there are *multiple* libraries that give us this superpower. We're using Symfony's BrowserKit client... mostly because it's built into Symfony and easy to start using. But, there are others. My favorite is called Mink, which is used behind the scenes with Behat. We have a tutorial all about Behat, with big sections devoted to Mink. So, check that out.

Go Deeper!

Learn all about Mink and Behat: <https://knpuniversity.com/screencast/behata>

So... what's the difference between Symfony's BrowserKit and Mink? With BrowserKit, you're not *actually* making a real HTTP request to your app. Nope, you're making a "fake" request directly into your code. Honestly, that doesn't matter much. And actually, it makes setup a bit easier: we didn't need to configure that our site lived at `http://localhost:8000` or anything like that.

But, BrowserKit has one big disadvantage: you can't test JavaScript functionality. Nope, since these are fake requests, your JavaScript simply doesn't run! This is the main reason why I prefer Mink: it *does* allow you to run your code in a real browser... *with* JavaScript.

For the next few chapters, we *are* going to use Symfony's BrowserKit Client. But, most of the concepts transfer well to other test clients, like Mink. And you'll still be able to use most of the magic from LiipFunctionalTestBundle. If you have questions about this, just ask in the comments!

Next, let's talk about this `$crawler` object and how we can use it to dive into the HTML of the page!

Chapter 24: DomCrawler: Epic Beast of the Night

Our test is *already* pretty cool. This is called a smoke test: with just a *tiny* bit of code, we're at *least* making sure the homepage doesn't have a huge error!

And now... stop! Before we write any more tests, we need to ask ourselves a question: does the feature we're building *need* a test? We're going to add a list of all of the enclosures on the homepage. That doesn't sound too scary, so I might not test this in the real world. We *will* test it... yea know... because this is a tutorial on testing. But my point is: think before you test!

Writing the Enclosure Test

Ok, back to TDD! Our homepage is blank... but *soon* it will have a list of all of the enclosures in the database!

In the test, we need to think about how this might look. Let's add `$table = $crawler->filter()` and then a CSS selector: `.table-enclosures`. I'm saying that, when we build this page, we should create an element with this class. The `$crawler` object is a bit similar to the jQuery function in JavaScript: by using its `filter()` method, it's *really* good at finding elements via CSS. The `$table` variable is *itself* another Crawler object, which represents the table element.

```
21 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 6
7  class DefaultControllerTest extends WebTestCase
8  {
9      public function testEnclosuresAreShownOnHomepage()
10     {
... lines 11 - 16
17         $table = $crawler->filter('.table-enclosures');
... line 18
19     }
20 }
```

Now, we can `assertCount()` that 3 is equal to `$table->filter('tbody tr')`. In other words, inside the table, we expect there to be 3 rows. Why 3? Well... I just made that up! Just like with integration tests, we're going to need to take control of the database so that we know *exactly* what's inside. More on that soon!

```
21 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 8
9      public function testEnclosuresAreShownOnHomepage()
10     {
... lines 11 - 17
18         $this->assertCount(3, $table->filter('tbody tr'));
19     }
... lines 20 - 21
```

Try the test now:

```
$ ./vendor/bin/phpunit tests/AppBundle/Controller/DefaultControllerTest.php
```

Yay! It fails! That means we are ready to code!

Coding up the Feature

Open `DefaultController` and query the database for all the enclosures: `$this->getDoctrine()->getRepository(Enclosure::class)->findAll()`. Use this to pass a new enclosures variable into Twig.

19 lines | [app/Resources/views/default/index.html.twig](#)

... lines 1 - 2

```
3 {% block body %}
4     <h3>Enclosures</h3>
5
6     <table class="table-enclosures">
7         <tbody>
8
9
10
11
12
13
14
15
16     </tbody>
17 </table>
18 {% endblock %}
```

Now open that template! It's in `app/Resources/views/default/index.html.twig`. I'll add an `h3`, the table with `class="table-enclosures"` and a `tbody`.

19 lines | [app/Resources/views/default/index.html.twig](#)

... lines 1 - 2

```
3 {% block body %}
4     <h3>Enclosures</h3>
5
6     <table class="table-enclosures">
7         <tbody>
8
9
10
11
12
13
14
15
16     </tbody>
17 </table>
18 {% endblock %}
```

Inside, start looping! for enclosure in enclosures. For the `<tr>`, I'm going to give each a unique id. This will help us write a *different* test in a few minutes.

19 lines | [app/Resources/views/default/index.html.twig](#)

... lines 1 - 5

```
6     <table class="table-enclosures">
7         <tbody>
8             {% for enclosure in enclosures %}
9                 <tr id="enclosure-{{ enclosure.id }}">
10
11
12
13
14             </tr>
15             {% endfor %}
16         </tbody>
17     </table>
18
19 ... lines 18 - 19
```

And now... let's print some stuff! Like the Enclosure #, and on the next column, "Contains" then `enclosure.dinosaurCount` dinosaurs. Rawr!

19 lines | [app/Resources/views/default/index.html.twig](#)

... lines 1 - 7

```
8         {% for enclosure in enclosures %}
9             <tr id="enclosure-{{ enclosure.id }}">
10                 <td>Enclosure #{{ enclosure.id }}</td>
11                 <td>
12                     Contains <strong>{{ enclosure.dinosaurCount }}</strong> dinosaur(s)
13                 </td>
14             </tr>
15         {% endfor %}
```

... lines 16 - 19

PhpStorm is angry... and it's right! We don't have `getId()` or `getDinosaurCount()` methods yet.

Open up `Enclosure`. Near the top, at `getId()`: it should return a nullable int. And at the bottom, create a public function `getDinosaurCount()` that will return an int. Return `$this->dinosaurs->count()`.

100 lines | [src/AppBundle/Entity/Enclosure.php](#)

... lines 1 - 14

```
15 class Enclosure
```

```
16 {
```

... lines 17 - 44

```
45     public function getId(): ?int
```

```
46     {
```

```
47         return $this->id;
```

```
48     }
```

... lines 49 - 94

```
95     public function getDinosaurCount(): int
```

```
96     {
```

```
97         return $this->dinosaurs->count();
```

```
98     }
```

```
99 }
```

So... does the page work? It should! Try the tests!



```
$ ./vendor/bin/phpunit tests/AppBundle/Controller/DefaultControllerTest.php
```

Ah! Failure because the actual size one does not match expected size 3. Check out the homepage in the browser: we have 7 enclosures! This is the same problem we had with integration test: we're not taking control of the data in the database. So... we really have *no* idea how many enclosures will be in the list! Our main database has 7... and apparently our test database has only one.

Let's fix this!

Chapter 25: Test Fixtures & Fast Databases!

In practice, the *hardest* thing about functional testing isn't all the stuff about clicking links or filling out forms. Nope, the toughest thing is taking control of the database!

[To Fixture or Not to Fixture?](#)

To do this, there are two big philosophies! First, like with integration tests, we can decide to always start the database empty. And then, if we need data - like we need to add some Enclosures to the database - we will add that data *inside* the test method itself. This is how I normally code. It's not super fancy, and it means that you need to do extra work in each test to create the exact data you need. But, it *also* means that each test reads like a complete story. For example, at the top of this test, you would be able to see that we created 3 Enclosure objects. Then, at the bottom, it will make sense *why* we're *expecting* to see 3 rows in the table.

The second philosophy, which is a bit simpler, is to load data fixtures. This is what we're going to do: but I'll mention how things would be different if you want to use the first philosophy.

[Adding Data Fixtures](#)

First, install the DoctrineFixturesBundle:

```
$ composer require --dev doctrine/doctrine-fixtures-bundle:2.4.1
```

If you downloaded the course code, in the tutorial/ directory, you should have a DataFixtures directory. Copy that into your AppBundle.

```
54 lines | src/AppBundle/DataFixtures/ORM/LoadBasicParkData.php
... lines 1 - 10
11 class LoadBasicParkData extends AbstractFixture implements OrderedFixtureInterface
12 {
13     public function load(ObjectManager $manager)
14     {
15         $carnivorousEnclosure = new Enclosure();
16         $manager->persist($carnivorousEnclosure);
17         $this->addReference('carnivorous-enclosure', $carnivorousEnclosure);
18
19         $herbivorousEnclosure = new Enclosure();
20         $manager->persist($herbivorousEnclosure);
21         $this->addReference('herbivorous-enclosure', $herbivorousEnclosure);
22
23         $manager->persist(new Enclosure(true));
24
25         $this->addDinosaur($manager, $carnivorousEnclosure, 'Velociraptor', true, 3);
26         $this->addDinosaur($manager, $carnivorousEnclosure, 'Velociraptor', true, 1);
27         $this->addDinosaur($manager, $carnivorousEnclosure, 'Velociraptor', true, 5);
28
29         $this->addDinosaur($manager, $herbivorousEnclosure, 'Triceratops', false, 7);
30
31         $manager->flush();
32     }
33 ... lines 33 - 52
53 }
```

```

41 lines | src/AppBundle/DataFixtures/ORM/LoadSecurityData.php
... lines 1 - 10
11 class LoadSecurityData extends AbstractFixture implements OrderedFixtureInterface
12 {
13     public function load(ObjectManager $manager)
14     {
15         $herbivorousEnclosure = $this->getReference('herbivorous-enclosure');
16
17         $this->addSecurity($herbivorousEnclosure, 'Fence', true);
18
19         $carnivorousEnclosure = $this->getReference('carnivorous-enclosure');
20
21         $this->addSecurity($carnivorousEnclosure, 'Electric fence', false);
22         $this->addSecurity($carnivorousEnclosure, 'Guard tower', false);
23
24         $manager->flush();
25     }
... lines 26 - 39
40 }

```

These two classes build 3 Enclosures and also add some security to them. But, part of this code is using a `setEnclosure()` method on `Dinosaur`... and that doesn't exist! Open `Dinosaur`, scroll to the bottom, and add it: `public function setEnclosure()` with an `Enclosure` argument. Set that on the property.

```

87 lines | src/AppBundle/Entity/Dinosaur.php
... lines 1 - 10
11 class Dinosaur
12 {
... lines 13 - 81
82     public function setEnclosure(Enclosure $enclosure)
83     {
84         $this->enclosure = $enclosure;
85     }
86 }

```

Awesome! Once the bundle finishes downloading open `AppKernel`. And inside the if statement, add new `DoctrineFixturesBundle()`. If you're using `Flex`, this step will have already been done for you automatically.

```

62 lines | app/AppKernel.php
... lines 1 - 7
8 class AppKernel extends Kernel
9 {
10     public function registerBundles()
11     {
... lines 12 - 22
23         if (in_array($this->getEnvironment(), ['dev', 'test'], true)) {
... lines 24 - 26
27             $bundles[] = new DoctrineFixturesBundle();
... lines 28 - 36
37         }
... lines 38 - 39
40     }
... lines 41 - 60
61 }

```

We haven't hooked the fixtures into our tests yet, but we can at least try them! Run:

```
$ php bin/console doctrine:fixtures:load
```

Go check out the browser! Yes! The fixtures gave us 3 enclosures. That's why I wrote our test to expect 3 rows. If we can load the fixtures when the test runs, we're in business!

Loading Fixtures in the Test

Fortunately, LiipFunctionalTestBundle gives us a really nice way to do this. At the top of the test method, add `$this->loadFixtures()` and pass an array of the fixture classes you want to load: `LoadBasicParkData::class` and `LoadSecurityData::class`.

```
28 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 8
9  class DefaultControllerTest extends WebTestCase
10 {
11     public function testEnclosuresAreShownOnHomepage()
12     {
13         $this->loadFixtures([
14             LoadBasicParkData::class,
15             LoadSecurityData::class,
16         ]);
... lines 17 - 25
26     }
27 }
```

Tip

Since LiipFunctionalTestBundle v3.0 the `loadFixtures()` method is no longer supported. You should use [LiipTestFixturesBundle](#) instead

If you're going to use the same set of fixtures for all your test methods, then moving this to `setUp()` is a great choice.

Run the tests!

```
$ ./vendor/bin/phpunit tests/AppBundle/Controller/DefaultControllerTest.php
```

They work! They pass, over and over again!

So, how would things be different if you did *not* want to load fixtures? Well, you *will* still want to empty the database. So, you could use the same trick as the integration tests. Or, you could call `$this->loadFixtures()` with an empty array.

Of course, the tests fail. That's because `loadFixtures()` empties the database... but then doesn't load *anything* into it.

Remember, if you choose this philosophy, you're now responsible for creating the data you need. How? The same way you always do: create some Enclosure objects and then persist them with the EntityManager. And since we're *still* ultimately extending `KernelTestCase`, we already know how to get the EntityManager: with `self::$kernel->getContainer()->get('doctrine')->getManager()`.

Test Base Classes

Actually, it would be *great* if we had a shortcut like `$this->getEntityManager()` for *all* our test classes. We won't do it in this tutorial, but I *highly* recommend creating your *own* base test class with extra shortcut methods. Typically, I'll have one base test class for integration tests - which extends `KernelTestCase` - and if necessary, another for my functional tests, which extends `WebTestCase`. You can also use traits to share code even better.

Faster Database Loading

The LiipFunctionalTestBundle has two other tricks. First, if you're using SQLite, then *it* automatically builds the schema for

you. Check this out: delete the database file:

```
$ rm var/data/test.sqlite
```

Bye bye database schema! But, when you run the tests, they still pass! When you load the fixtures, it creates the schema too. Thanks friends!

The second trick lives in `app/config/config_test.yml`. Add a new option: `cache_sqlite_db` set to `true`.

```
30 lines | app/config/config_test.yml
... lines 1 - 27
28 liip_functional_test:
29   cache_sqlite_db: true
```

Visually... this doesn't make any difference. BUT! Behind the scenes, cool things are happening. Each time you call `loadFixtures()`, it loads the fixtures and then *caches* the database file. The next time you call `loadFixtures()` with the same arguments, it instantly re-uses that cached database file.

Check this out: to simulate loading a lot of fixtures, add a `sleep(5)` in one of them. Now, run the test:

```
$ ./vendor/bin/phpunit tests/AppBundle/Controller/DefaultControllerTest.php
```

Yea... it's *slow*. The bundle detected the change we made and was smart enough to know that it needed to reload the fixtures. But the second time... zoom! It's super fast.

The *coolest* part is that all of this database and fixture-handling stuff from `LiipFunctionalTestBundle` can be used even if you decide to use a different client - like `Mink` - instead of `Symfony's BrowserKit`.

Next, let's look at one more trick you can do with fixtures.

Chapter 26: Loading Fixtures References

Despite all our precautions, we still *sometimes* have enclosures with *no* security. Yea... a lot of people are getting eaten, a lot of lawsuits - very expensive. To help with this, I want to add an "Alarm" button on the homepage next to any enclosures that do *not* have active security.

Because this sounds pretty important, let's write the test first. Add

public function testThatThereIsAnAlarmButtonWithoutSecurity(). Copy the fixture and request code from before and paste it here. But, at the end of loadFixtures(), add getReferenceRepository() and assign this to a new \$fixtures variable.

```
45 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 8
9  class DefaultControllerTest extends WebTestCase
10 {
... lines 11 - 27
28  public function testThatThereIsAnAlarmButtonWithoutSecurity()
29  {
30      $fixtures = $this->loadFixtures([
31          LoadBasicParkData::class,
32          LoadSecurityData::class,
33      ]->getReferenceRepository());
... lines 34 - 42
43  }
44 }
```

[What are Fixture References?](#)

Here's the deal: if you look in the fixtures, you can see that the first two Enclosures do *not* have any security. You can *also* see that we're using some sort of "reference" system. This allows us to store a specific object in memory so that we can re-use it somewhere else. For example, in LoadSecurityData, we get the herbivorous-enclosure object out and *add* security! We're safe from those wild veggie eating dinos!

It does the same for carnivorous-enclosure... but then adds two Security objects that are both *inactive*. Doh! Yep, this means that the carnivorous-enclosure is the only Enclosure that is *not* secure. In the test, our goal is to assert that, on the homepage, *this* exact Enclosure has the alarm button.

And we planned ahead for this! Remember, in the template, we added an enclosure-`{id}` to each tr element. So if we can get the actual id value of the Carnivorous Enclosure, it will be *really* easy to find its tr element and look for the alarm button. The reference system gives us that power!

Yep, we can fetch the exact Enclosure object by saying `$enclosure = $fixtures->getReference('carnivorous-enclosure')`. Next, create a `$selector` variable set to `sprintf('#enclosure-%s .button-alarm')` and `$enclosure->getId()`. We'll expect the alarm button to have this class.

```

45 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 8
9 class DefaultControllerTest extends WebTestCase
10 {
... lines 11 - 27
28 public function testThatThereIsAnAlarmButtonWithoutSecurity()
29 {
... lines 30 - 38
39 $enclosure = $fixtures->getReference('carnivorous-enclosure');
40 $selector = sprintf('#enclosure-%s .button-alarm', $enclosure->getId());
... lines 41 - 42
43 }
44 }

```

Finish the test! `$this->assertGreaterThan(0, $crawler->filter($selector)->count())`.

```

45 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 8
9 class DefaultControllerTest extends WebTestCase
10 {
... lines 11 - 27
28 public function testThatThereIsAnAlarmButtonWithoutSecurity()
29 {
... lines 30 - 41
42 $this->assertGreaterThan(0, $crawler->filter($selector)->count());
43 }
44 }

```

I love it! So first, of course, make sure the test fails. Copy the method name and run phpunit with the `--filter` option:

```
$ ./vendor/bin/phpunit --filter testThatThereIsAnAlarmButtonWithoutSecurity
```

Awesome!

Adding the Alarm Button

So let's code! In `index.html.twig`, add one more `<td>`: if `enclosure.isSecurityActive()` with `else` and `endif`.

```

16 lines | app/Resources/views/default/index.html.twig
... lines 1 - 7
8 {% for enclosure in enclosures %}
9 <tr id="enclosure-{{ enclosure.id }}">
... lines 10 - 13
14 <td>
15 {% if enclosure.isSecurityActive %}

```

If security *is* active, we rock! Add a cute lock icon and say "Security active". Yep, just sit back and enjoy some Jolt soda: nobody is getting eaten today!

```

16 lines | app/Resources/views/default/index.html.twig
... lines 1 - 13
14 <td>
15 {% if enclosure.isSecurityActive %}
16

```

But if security is *not* active, ah crap! Add the button with the button-alarm class that the test is looking for. And say "Sound alarm!!!".

```
16 lines | app/Resources/views/default/index.html.twig
... lines 1 - 13
14          <td>
... lines 15 - 16
```

That should be it! Run the test:

```
$ ./vendor/bin/phpunit --filter testThatThereIsAnAlarmButtonWithoutSecurity
```

Ha! It passes!

[Debugging Functional Tests](#)

But... what if it *didn't* pass? Well... the errors wouldn't be very helpful: it would basically just say that 0 is not greater than 0. When things fail, the trick is to go *above* the failure and `dump($client->getResponse()->getContent())`. If you're using Flex, make sure to install the var-dumper package.

Now when you run the test, it will *at least* print out the HTML body. By the way, with a little bit of clever coding, you can hook into the `onNotSuccessfulTest` method and have the last response content printed automatically when a test fails. I'll leave that as a challenge for you. But, ask us in the comments if you have any questions.

Ok, there's *one* more thing I want to talk about with functional tests: filling out and submitting a form.

Chapter 27: Testing a Form Submit

New feature request! On the homepage, management wants a form where they can choose an enclosure, write a dinosaur spec - like "Large herbivore" and submit! Behind the scenes, we will create that new Dinosaur and put it into the Enclosure.

Since we're now functional-testing pros, let's get right to the test! Add public function `testItGrowsADinosaurFromSpecification()`. And as usual, steal some code from earlier and paste it on top. You can start to see how some of this could be refactored to a `setUp` method.

```
59 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 8
9  class DefaultControllerTest extends WebTestCase
10 {
... lines 11 - 44
45  public function testItGrowsADinosaurFromSpecification()
46  {
47      $this->loadFixtures([
48          LoadBasicParkData::class,
49          LoadSecurityData::class,
50      ]);
51
52      $client = $this->makeClient();
53
54      $crawler = $client->request('GET', '/');
55
56      $this->assertStatusCode(200, $client);
57  }
58 }
```

After creating the client, add `$client->followRedirects()`. Normally, when our app redirects, Symfony's Client does *not* follow the redirect. Sometimes that's useful... but this line makes it behave like a normal browser.

```
60 lines | tests/AppBundle/Controller/DefaultControllerTest.php
... lines 1 - 44
45  public function testItGrowsADinosaurFromSpecification()
46  {
... lines 47 - 52
53  $client->followRedirects();
... lines 54 - 57
58  }
... lines 59 - 60
```

[Filling in the Form Fields](#)

To fill out the form fields, first we need to *find* the form. Do that with `$form = $crawler->selectButton()` and pass this the *value* of the button that will be on your form. How about "Grow dinosaur". Then call `->form()`.

62 lines | tests/AppBundle/Controller/DefaultControllerTest.php

... lines 1 - 44

```
45     public function testItGrowsADinosaurFromSpecification()
46     {
    ... lines 47 - 58
59     $form = $crawler->selectButton('Grow dinosaur')->form();
60     }
    ... lines 61 - 62
```

We now have a Form object. No, not Symfony's normal Form object from its form system. This is from the DomCrawler component and its job is to help us fill out its fields.

So let's think about it: we will need 2 fields: an enclosure select field and a specification text box. To fill in the first, use `$form['enclosure']` - the enclosure part is whatever the name attribute for your field will be. If you're using Symfony forms, usually this will look more like `dinosaur[enclosure]`.

Then, because this will be a select field, use `->select(3)`, where 3 is the value of the option element you want to select. Do this again for a specification field. Setting this one is easier: `->setValue('large herbivore')`.

64 lines | tests/AppBundle/Controller/DefaultControllerTest.php

... lines 1 - 44

```
45     public function testItGrowsADinosaurFromSpecification()
46     {
    ... lines 47 - 59
60     $form['enclosure']->select(3);
61     $form['specification']->setValue('large herbivore');
62     }
    ... lines 63 - 64
```

Honestly, I don't love Symfony's API for filling in forms - I like Mink's better. But, it works fine. When the form is ready, submit with `$client->submit($form)`. That will submit to the correct URL and send all the data up!

70 lines | tests/AppBundle/Controller/DefaultControllerTest.php

... lines 1 - 44

```
45     public function testItGrowsADinosaurFromSpecification()
46     {
    ... lines 47 - 62
63     $client->submit($form);
    ... lines 64 - 67
68     }
    ... lines 69 - 70
```

But... now what? What should the user see after submitting the form? Well... we should probably redirect *back* to the homepage with a nice message explaining what just happened. Use `$this->assertContains()` to look for the text "Grew a large herbivore in enclosure #3" inside `$client->getResponse()->getContent()`.

70 lines | tests/AppBundle/Controller/DefaultControllerTest.php

... lines 1 - 44

```
45     public function testItGrowsADinosaurFromSpecification()
46     {
    ... lines 47 - 63
64     $this->assertContains(
65         'Grew a large herbivore in enclosure #3',
66         $client->getResponse()->getContent()
67     );
68     }
    ... lines 69 - 70
```

Test, done! Copy the method name and *just* run this test:

```
$ ./vendor/bin/phpunit --filter testItGrowsADinosaurFromSpecification
```

Perfect! It fails with

The current node list is empty.

This is a really common error... though it's not the *most* helpful. It basically means that some element could not be found.

Code the Form

With the test done, let's code! And... yea, let's take a shortcut! In the tutorial/ directory, find the app/Resources/views/_partials folder, copy it, and paste it in *our* app/Resources/views directory.

23 lines | tutorial/app/Resources/views/_partials/ newDinoForm.html.twig

```
1  <form action="{{ url('grow_dinosaur') }}" method="POST">
2    <div class="row">
3      <div class="column">
4        <label for="enclosure">Enclosure</label>
5        <select name="enclosure" id="enclosure">
6          {% for enclosure in enclosures %}
7            <option value="{{ enclosure.id }}">Enclosure #{{ enclosure.id }}</option>
8          {% endfor %}
9        </select>
10     </div>
11
12     <div class="column">
13       <label for="specification">Dino description</label>
14       <input type="text" id="specification" name="specification" placeholder="Small carnivorous dino friend" />
15     </div>
16
17     <div class="column">
18       <label for="">&nbsp;</label>
19       <input type="submit" class="button" value="Grow dinosaur" />
20     </div>
21   </div>
22 </form>
```

Then, at the top of index.html.twig, use it: include('_partials/_newDinoForm.html.twig').

56 lines | [src/AppBundle/Controller/DefaultController.php](#)

... lines 1 - 5

```
6 use AppBundle\Factory\DinosaurFactory;
7 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
... lines 8 - 11
12 class DefaultController extends Controller
13 {
... lines 14 - 27
28 /**
29  * @Route("/grow", name="grow_dinosaur")
30  * @Method({"POST"})
31  */
32 public function growAction(Request $request, DinosaurFactory $dinosaurFactory)
33 {
34     $manager = $this->getDoctrine()->getManager();
35
36     $enclosure = $manager->getRepository(Enclosure::class)
37         ->find($request->request->get('enclosure'));
38
39     $specification = $request->request->get('specification');
40     $dinosaur = $dinosaurFactory->growFromSpecification($specification);
41
42     $dinosaur->setEnclosure($enclosure);
43     $enclosure->addDinosaur($dinosaur);
44
45     $manager->flush();
46
47     $this->addFlash('success', sprintf(
48         'Grew a %s in enclosure #%d',
49         mb_strtolower($specification),
50         $enclosure->getId()
51     ));
52
53     return $this->redirectToRoute('homepage');
54 }
55 }
```

The form is really simple: it's not even using Symfony's form system! You can see the name="enclosure" select field where the value for each option is the enclosure's id. Below that is the name="specification" text field and the "Grow dinosaur" button the test relies on.

For the submit logic, go back into the tutorial/ directory, find DefaultController and copy all of the growAction() method. Paste this into *our* DefaultController. Oh, and we need a few use statements: re-type part of @Method and hit tab to add its use statement. Do the same for DinosaurFactory.

56 lines | [src/AppBundle/Controller/DefaultController.php](#)

... lines 1 - 5

```
6 use AppBundle\Factory\DinosaurFactory;
7 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
... lines 8 - 11
12 class DefaultController extends Controller
13 {
... lines 14 - 27
28 /**
29  * @Route("/grow", name="grow_dinosaur")
30  * @Method({"POST"})
31  */
32 public function growAction(Request $request, DinosaurFactory $dinosaurFactory)
33 {
34     $manager = $this->getDoctrine()->getManager();
35
36     $enclosure = $manager->getRepository(Enclosure::class)
37         ->find($request->request->get('enclosure'));
38
39     $specification = $request->request->get('specification');
40     $dinosaur = $dinosaurFactory->growFromSpecification($specification);
41
42     $dinosaur->setEnclosure($enclosure);
43     $enclosure->addDinosaur($dinosaur);
44
45     $manager->flush();
46
47     $this->addFlash('success', sprintf(
48         'Grew a %s in enclosure #%d',
49         mb_strtolower($specification),
50         $enclosure->getId()
51     ));
52
53     return $this->redirectToRoute('homepage');
54 }
55 }
```

Ok, it's happy! Sure, the code is lacking the normal security and safeguards we expect when using Symfony's form system... but it's only a dinosaur park people! We *do*, however, have the success flash message!

So if we haven't messed anything up, it should work. Try the test!



```
$ ./vendor/bin/phpunit --filter testItGrowsADinosaurFromSpecification
```

Yes! It passes! We just confirmed that this form works before we ever even loaded it in a browser. That's pretty cool.

So that's the power of functional tests. And I find them *especially* powerful when using Mink and testing that my JavaScript works.

Ok guys, just *one* more topic left, and it's *fun*! Continuous integration! You know, the fancy term that means: let the robots run your tests automatically!

Chapter 28: Continuous Integration: Activate the Robots

Great news! We have a growing test suite! Bad news! Nobody *actually* likes to run the tests! Sure, they only take 4 seconds now. But in a real app - with integration tests and functional tests - the entire suite may take 5 minutes, or 10 minutes... or an hour. No joke!

In practice, I *only* run the specific tests that I'm currently working with. And that's fine! Because we at KnpU use continuous integration. It's a *must*. Yep, each time we push to any branch, a bunch of friendly robots checkout our code, run our tests, and report back. This keeps *us* coding, and we learn about failures as quickly as possible.

Oh, and continuous integration can do other cool stuff too... like deploy!

[Hello CircleCI](#)

There are a few tools for continuous integration. If you want to host it yourself, Jenkins is the way to go. But if you want easy setup, you can use Travis CI or our favorite, CircleCI!

And actually, CircleCI even has a free plan. Hey CircleCI - that's *super* nice of you!

Before we setup our project, we need to host our code somewhere. Head over to GitHub and create a new repository, give it a name, and submit. Copy the 2 lines to push to an existing repository and move over to your terminal. Make sure to commit all your changes so far - I already did that. Then, paste!

Back on the browser... we have a repo!

[CircleCI Initial Setup](#)

On CircleCI, make sure you've got your personal organization selected in the upper left. Then, go to projects, "Add Project" and find the new repository. If you don't see it, try refreshing. There it is! Click "Setup Project".

Awesome! We *are* going to use version 2.0 of their platform: it's all based on Docker and is super trendy. PHP is already selected as the language, so we can just click the "Copy to clipboard" button.

Basically, the *only* thing we need to do is create a config.yml file with instructions on how to build our project: what container image to use, what tools we need installed, and what commands run the tests. CircleCI summons the robot army and takes care of the rest.

[Bootstrapping .circleci/config.yml](#)

Over in our editor, create a .circleci directory with a config.yml file inside. Paste!

```

38 lines | .circleci/config.yml
1  # PHP CircleCI 2.0 configuration file
2  #
3  # Check https://circleci.com/docs/2.0/language-php/ for more details
4  #
5  version: 2
6  jobs:
7    build:
8      docker:
9        # specify the version you desire here
10       - image: circleci/php:7.1.5-browsers
11
12       # Specify service dependencies here if necessary
13       # CircleCI maintains a library of pre-built images
14       # documented at https://circleci.com/docs/2.0/circleci-images/
15       # - image: circleci/mysql:9.4
16
17     working_directory: ~/repo
18
19     steps:
20       - checkout
21
22       # Download and cache dependencies
23       - restore_cache:
24         keys:
25           - v1-dependencies-{{ checksum "composer.json" }}
26           # fallback to using the latest cache if no exact match is found
27           - v1-dependencies-
28
29       - run: composer install -n --prefer-dist
30
31       - save_cache:
32         paths:
33           - ./vendor
34         key: v1-dependencies-{{ checksum "composer.json" }}
35
36       # run tests!
37       - run: phpunit

```

Wow! This is great! CircleCI is *super* powerful... but it can be a little confusing at first. That's probably why they gave us this amazing starter config! It uses a recent PHP image. The `-browsers` part means it comes pre-installed with tools that help you test in a browser... like if you're using Mink. That's awesome!

It also installs the composer dependencies, does some smart caching to speed up builds, and executes PHPUnit. Actually, that's the only thing I want to change right now: use `./vendor/bin/phpunit` instead of the globally installed version.

```

38 lines | .circleci/config.yml
... lines 1 - 5
6  jobs:
7    build:
... lines 8 - 18
19   steps:
... lines 20 - 36
37   - run: ./vendor/bin/phpunit

```

Let's see what happens! In the terminal, add the directory, create a commit, and push!

```
$ git add .circleci
$ git commit -m "adding CircleCI config - Woo"
$ git push origin master
```

This won't activate our first build *yet*, but we *do* have a functional config file. And that means we can click "Start Building". Deep breath. Do it!

This installs a webhook on GitHub so that every push will automatically trigger a build. And because CircleCI is so friendly, it even started our first build.

[Where is Composer?](#)

And... within 2 seconds, it failed! Geez! It *did* start the environment, which means it built the container image. Sometimes the image will be cached - it was this time. When that happens, it's *super* fast. Then, it checked out our code and... huh! It failed because composer is missing?

That's especially weird because this is *their* default config! The fix is mysterious. Ready for it? Remove the .5 from the image name.

```
38 lines | .circleci/config.yml
... lines 1 - 5
6 jobs:
7   build:
8     docker:
... line 9
10     - image: circleci/php:7.1-browsers
... lines 11 - 38
```

In your terminal, commit this - "Using a different image" and push to origin master:

```
$ git add -u
$ git commit -m "Using a different image"
$ git push origin master
```

This *should* trigger build #2. Click on master and... there it is! So... why did we do that? It's actually a bug with CircleCI. For some reason, *some* of their images are missing composer. From a bit of debugging I did earlier, I found out that the 7.1 image *has* composer, but the 7.1.5 is missing it. Weird!

Anyways, this time you can see it build the image: it was *not* cached. And awesome! It's downloading our composer deps! And saving cache and... woh!

PHPUnit ran! And... it *almost* passed. The failures are all the same:

```
Unable to open database file
```

Of course! Remember: our test database is stored in var/data. But since none of this is committed, there *is* no var/data directory. That means that sqlite file can't be created.

For better or worse, this is the flow when you setup continuous integration: make some tweaks, push, wait, debug and repeat. But! If you have a *really* tricky problem, you can actually "rebuild with SSH access". This runs the build again, but then keeps the container alive after and gives you SSH access. That's an *amazing* way to run some commands and find out what's going wrong.

Back in config.yml, add a new run line: mkdir var/data.

40 lines | [.circleci/config.yml](#)

... lines 1 - 5

6 jobs:

7 build:

... lines 8 - 18

19 steps:

... lines 20 - 35

36 - run: mkdir var/data

... lines 37 - 40

You know the drill: find your terminal, commit that change and push!

[CircleCI Reference](#)

We're not going to go into deep detail about the CircleCI config. But, if you Google for "CircleCI Config Reference", you'll find an [awesome page](#) on their documentation.

And woh! CircleCI is *powerful*. In our [Ansistrano](#) tutorial, we used CircleCI and workflows to *deploy* our code. The tests would run first, and *if* they passed, it would trigger a second "deploy" job. Cool stuff!

Go back to the CircleCI page and find the latest build. Ha! It *passed*!

In about five minutes, our project has continuous integration!

Next, let's look at one more cool thing with CircleCI: artifacts... which are test-debugging gold.

Chapter 29: CircleCI Artifacts

The *hardest* thing about CI is when tests fail on CircleCI, but pass locally. When you have a lot of functional tests, this will happen more often than you think. Usually, it's a timing issue: you click to open a JavaScript modal and then click a link *in* that modal. This works locally... but for some reason that modal loads a *little* bit slower in CircleCI. Your test fails because you try to click the link too quickly. We talk about this in detail in our [Behat tutorial](#).

The "Rebuild with SSH" option is *great* for this. But an even *better* tool is artifacts. Very simply, artifacts are a way for you to save data - like logs or browser screenshots when a test fails - and make it accessible from the web interface. Imagine seeing 4 test failures and seeing 4 *screenshots* right on the UI of what the browser looked like the moment those tests failed! We actually have a [blog post](#) about getting this setup. That post uses CircleCI version 1.0 - but once we talk about artifacts, you should be able to translate to version 2 without a problem.

Artifacts in Action

Let's see artifacts in action. Go back to the config file. At the end of the phpunit command add `--log-junit ~/phpunit/junit.xml`.

```
46 lines | .circleci/config.yml
... lines 1 - 5
6 jobs:
7   build:
... lines 8 - 18
19   steps:
... lines 20 - 38
39     - run: ./vendor/bin/phpunit --log-junit ~/phpunit/junit.xml
... lines 40 - 46
```

This flag tells PHPUnit to output some logs in a standard "JUnit" format. This is basically a detailed diagnostic of what happened during the tests.

Now, add two more steps: `store_test_results` with a path option set to `~/phpunit`. And another one called `store_artifacts` with that same option.

```
46 lines | .circleci/config.yml
... lines 1 - 5
6 jobs:
7   build:
... lines 8 - 18
19   steps:
... lines 20 - 40
41     - store_test_results:
42       path: ~/phpunit
... line 43
44     - store_artifacts:
45       path: ~/phpunit
```

Let's commit this first, trigger the build, and *then* talk. Commit wildly... then push!

Understanding CircleCI Steps

A CircleCI build consists of these "steps", and each step uses a built-in step "type". The most common and useful type is `run`. But we're also using `checkout` near the top and now `store_test_results` and `store_artifacts`. These are all explained on that config reference page.

`store_artifacts` is the simpler of the two: whatever files we store as an artifact will become available for us to see & download via the web UI - or API. That means that we'll be able to see the `junit.xml` file... or any logs or screenshots we choose to store.

The `store_test_results` is also really cool. Thanks to this, CircleCI will parse the `junit.xml` file and *learn* things about your tests: like how *long* they took to run and their favorite color is... err... maybe not that one.

Anyways, let's go find our build! It passed! Wow, and it only took 48 seconds!

First, look under "Artifacts". Yea! Here is our `junit.xml` file. It's not super attractive, but you get the point. On KnpU, this is full of screenshots for any tests that failed.

Now, click on "Test Summary". Cool! 29 tests and 0 failures. And it even knows which tests are the *slowest*.

Go back and look at the previous build - build #3. The "Test Summary" part was *empty*! This data was filled in thanks to the `store_test_results` step. It's not mission-critical, but it's free functionality!

Other CI Uses

By the way, you can also use continuous integration for other things beyond tests, like enforcing code standards. By installing the [php-cs-fixer](#), you could easily make your build *fail* if someone pushes code that doesn't follow the standards.

And... we're done! Thanks for traveling along with me, dodging dinosaurs, and putting them back into their Enclosures. It's a thankless job, but somebody has to do it.

With testing as a part of your toolkit, your life will be *so* much better. Tests allow us to create features faster and code more aggressively. They give us the confidence that we're not going to break something important on the site. And you guys know me: I'm a pragmatist. I don't test for some philosophical reason about code quality. I test because it allows me to ship a high-quality app with *confidence*.

And besides, using continuous integration is *super* fun! If that's not reason enough to write tests, I don't know what is.

Ok guys, seeya next time!

