PHPUnit: Unit Testing with a Bite!

With <3 from SymfonyCasts

Chapter 1: PHPUnit Install

Hey everyone! Welcome to PHPUnit: testing with a bite! The tutorial where we discover, to our horror, that yet *another* Dinosaur theme park has built their systems... without any tests. It won't matter whether or not the raptors can open doors... if the fences never turn on.

Our park is called Dinotopia. And, to help wrangle our prehistoric friends, we've written a simple app that shows us which dinos are where and... how they're feeling. As you'll see, it's great! Except for the complete lack of tests.

App Setup

Anyways, to learn the most about testing and guarantee that nothing deadly will escape from *your* application, you should code along with me. After clicking "Download" on this page, unzip the file and move into the `start/` directory to find the code you see here. Check out the `README.md` for all the setup details.

The *last* step will be to open up a terminal and run:

```
symfony serve -d
```

to start a local web server on `127.0.0.1` port `8000`.

Cool! Move over to your browser, open a tab, go to `localhost:8000` ... and yes! Our Dinotopia Status app!

The App: Dinotopia Status

This simple app has the name of each dino, genus, size, and which enclosure the dino is currently hanging out in. Down here at the bottom, we also have a link to GenLab's super secret `dino-park` repository on GitHub. OoooO. This is where the engineers regularly post updates to let Bob, our resident park ranger, know which dinos are feeling good, need their medicine, or have escaped. Wait, What?! Hopefully, GitHub doesn't go offline when *that* happens.

And that's where we come in! We've already built the first version of the Dinotopia Status app. Looking at the code behind this, it's pretty simple: one controller

28 lines | src/Controller/MainController.php

```
... lines 1 - 2
namespace App\Controller;

use App\Entity\Dinosaur;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class MainController extends AbstractController
{
    #[Route(path: '/', name: 'main_controller', methods: ['GET'])]
    public function index(): Response
    {
        $dinos = [
            new Dinosaur('Daisy', 'Velociraptor', 2, 'Paddock A'),
            new Dinosaur('Maverick','Pterodactyl', 7, 'Aviary 1'),
            new Dinosaur('Big Eaty', 'Tyrannosaurus', 15, 'Paddock C'),
            new Dinosaur('Dennis', 'Dilophosaurus', 6, 'Paddock B'),
            new Dinosaur('Bumpy', 'Triceratops', 10, 'Paddock B'),
        ];

        return $this->render('main/index.html.twig', [
            'dinos' => $dinos,
        ]);
    }
}
```

one `Dinosaur` class...

40 lines | src/Entity/Dinosaur.php

```
... lines 1 - 2
namespace App\Entity;

class Dinosaur
{
    private string $name;
    private string $genus;
    private int $length;
    private string $enclosure;

    public function __construct(string $name, string $genus = 'Unknown', int $length = 0, string $enclosure = 'Unknown')
    {
        $this->name = $name;
        $this->genus = $genus;
        $this->length = $length;
        $this->enclosure = $enclosure;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function getGenus(): string
    {
        return $this->genus;
    }

    public function getLength(): int
    {
        return $this->length;
    }

    public function getEnclosure(): string
    {
        return $this->enclosure;
    }
}
```

and exactly *zero* tests. *Our* job is to fix that. We're also going to *add* a feature where we read each dino's status from GitHub and render it. That'll help Bob avoid going into the enclosure of Big Eaty - our resident T-Rex - when his status is "Hungry". Those accidents involve a *lot* of paperwork. And thanks to our tests, we'll ship that feature bug-free. You're welcome, Bob!

If you're new to testing, it can be intimidating. There are Unit tests, functional tests, integration tests, acceptance tests, math tests! The list is almost endless. We'll talk about all of these - except for math tests - throughout this series. In this tutorial, we're going to zoom in on unit tests: tests that cover one specific piece of code - like a function or method.

Oh, and by the way, tests are also *super* fun. It's automation! So buckley up.

Installing PHPUnit

What's the first step to writing tests? Installing PHP's defacto standard testing tool: PHPUnit. Move over to your terminal and run:

```
composer require --dev symfony/test-pack
```

This `test-pack` is a Symfony "pack" that will install PHPUnit - which is all we need right now - as well as some other libraries that'll come in handy later.

After it finishes, run:

```
git status
```

Cool! In addition to installing the packages,it looks like some Symfony Flex recipes modified and created a few other files. Ignore these for now. We'll talk about each one at some point in this series when they become relevant.

Ok, we're ready to write our *first* test! Let's do that next.

Chapter 2: Our First Test

We already have this `Dinosaur` class... and it's *pretty* simple. But when it comes to dinosaurs, bugs in our code can be, mmm, a bit painful. So let's add some basic tests!

Creating the Test Class

Mmmm... where do we put this new test? We can *technically* put our tests *anywhere* within our project. But when we installed `symfony/test-pack`, Flex created a `tests/` directory which, no surprise, is the *recommended* place to put our tests.

Remember that, in this tutorial, we're only dealing with Unit tests. So, inside of `tests/`, create a new directory called `Unit`. And because our `Dinosaur::class` lives in the `Entity` namespace - create an `Entity` directory inside of *that* at the same time.

All of this organization is *technically* optional: you can organize the `tests/` directory *however* you want. *But*, putting all of our unit tests into a `Unit` directory is just... nice. And the reason we made the `Entity` directory is because we want the file structure inside of `Unit` to mirror our `src/` directory structure. That's a best practice that keeps our tests organized.

Finally, create a new class called `DinosaurTest`. Using that `Test` suffix makes sense: we're testing `Dinosaur`, so we call this `DinosaurTest`! But it's also a requirement: PHPUnit - our testing library - *requires* this. It also requires that each class extend `TestCase`:

14 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 2
namespace App\Tests\Unit\Entity;

use PHPUnit\Framework\TestCase;

class DinosaurTest extends TestCase
{
... lines 9 - 12
}
```

Now let's go ahead and write a simple test to make sure everything is working.

Inside our `DinosaurTest` class, let's add `public function testIsWorks()` ... where we'll create the most *exciting* test ever! If you like return types - I do! - use `void` ... though that's optional

Inside call `self::assertEquals(42, 42)`:

14 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 2
namespace App\Tests\Unit\Entity;

use PHPUnit\Framework\TestCase;

class DinosaurTest extends TestCase
{
    public function testItWorks(): void
    {
        self::assertEquals(42, 42);
    }
}
```

That's it! It's not a very *interesting* test - if our computer thinks that 42 doesn't equal 42, we have bigger problems - but it's *enough*.

Executing PHPUnit

How do we *execute* the test? By executing PHPUnit. At your terminal, run:

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

And... awesome! PHPUnit saw *one* test - for our one test method - and one *assertion*.

We could also say `bin/phpunit` to execute our tests, which is basically just a shortcut to run `vendor/bin/phpunit`.

But, I'm sure your curious... What's... an assertion?

Looking back at `DinosaurTest`, the one assertion refers to the `assertEquals()` method, which comes from PHPUnit's `TestCase` class. If the *actual* value - 42 - doesn't match the *expected* value, the test would fail. PHPUnit has a *bunch* more assertion methods... and we can see them all by going to https://phpunit.readthedocs.io. This is *full* of goodies, including an "Assertions" section. And... *wow*! Look at them all... We'll talk about the most important assertions throughout the series. But for now, back to the test!

Test Naming Conventions

Because, I have a question: how did PHPUnit *know* that this is a test? When we call `vendor/bin/phpunit`, PHPUnit does three things. First, it looks for its configuration file, which is `phpunit.xml.dist`:

43 lines | phpunit.xml.dist

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- https://phpunit.readthedocs.io/en/latest/configuration.html -->
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="vendor/phpunit/phpunit/phpunit.xsd"
         backupGlobals="false"
         colors="true"
         bootstrap="tests/bootstrap.php"
         convertDeprecationsToExceptions="false"
>
    <php>
        <ini name="display_errors" value="1" />
        <ini name="error_reporting" value="-1" />
        <server name="APP_ENV" value="test" force="true" />
        <server name="SHELL_VERBOSITY" value="-1" />
        <server name="SYMFONY_PHPUNIT_REMOVE" value="" />
        <server name="SYMFONY_PHPUNIT_VERSION" value="9.5" />
    </php>

    <testsuites>
        <testsuite name="Project Test Suite">
            <directory>tests</directory>
        </testsuite>
    </testsuites>

    <coverage processUncoveredFiles="true">
        <include>
            <directory suffix=".php">src</directory>
        </include>
    </coverage>

    <listeners>
        <listener class="Symfony\Bridge\PhpUnit\SymfonyTestsListener" />
    </listeners>

    <!-- Run `composer require symfony/panther` before enabling this extension -->
    <!--
    <extensions>
        <extension class="Symfony\Component\Panther\ServerExtension" />
    </extensions>
    -->
</phpunit>
```

Inside, it finds `testsuites` ... and the `directory` part says:

43 lines | phpunit.xml.dist

```
... lines 1 - 3
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
... lines 5 - 9
>
... lines 11 - 19
    <testsuites>
        <testsuite name="Project Test Suite">
            <directory>tests</directory>
        </testsuite>
    </testsuites>
... lines 25 - 41
</phpunit>
```

> Hey PHPUnit: go look inside a `tests/` directory for tests!

Second, it finds that directory and *recursively* looks for every class that ends with the word `Test`. In this case, `DinosaurTest`. Finally, once it finds a test class, it gets a list of all of its public methods.

So... am I saying that PHPUnit will execute *every* public method as a test? Let's find out! Create a new `public function itWorksTheSame(): void`

19 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 6
7   class DinosaurTest extends TestCase
8   {
... lines 9 - 13
14      public function itWorksTheSame(): void
15      {
... line 16
17      }
18  }
```

Inside we are going to `self::assertSame()` that 42 is equal to 42. `assertSame()` is *very* similar to `assertEquals()` and we'll see the difference in a minute.

19 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 6
7   class DinosaurTest extends TestCase
8   {
... lines 9 - 13
14      public function itWorksTheSame(): void
15      {
16          self::assertSame(42, 42);
17      }
18  }
```

Now, move back to your terminal and let's run these tests again:

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

Huh? PHPUnit *still* says just one test and one assertion. But inside our test class, we have *two* tests and *two* assertions. The problem is that PHPUnit *only* executes public methods that are prefixed with the word `test`. You *could* put the `@test` annotation above the method, but that's not very common. So let's avoid being weird, and change this to `testItWorksTheSame()`.

19 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 6
7   class DinosaurTest extends TestCase
8   {
... lines 9 - 13
14      public function testItWorksTheSame(): void
15      {
16          self::assertSame(42, 42);
17      }
18  }
```

Now when we run the test:

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

PHPUnit sees 2 tests and 2 assertions! Shweeeet!

Testing Failures

What does it look like when a test fails? Let's find out! Change our expected `42` to a *string* inside `testItWorks()` ... and do the same inside `testItWorksTheSame()`. Yup, one of these *won't* work.

19 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 6
class DinosaurTest extends TestCase
{
    public function testItWorks(): void
    {
        self::assertEquals('42', 42);
    }

    public function testItWorksTheSame(): void
    {
        self::assertSame('42', 42);
    }
}
```

This time when we try it:

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

Oh no! One failure!

> `DinosaurTest::testItWorksTheSame()` failed asserting that `42` is identical to `42`.

So... `assertEquals()` *passed*, but `assertSame()` failed. That's because `assertEquals()` is the equivalent to doing an if `42 == 42`: using the double equal sign. But `assertSame()` is equivalent to `42 === 42`: with *three* equal signs.

And since the string 42 does *not* triple-equals the integer 42, that test fails and PHPUnit yells at us.

Ok, we've got our first tests behind us! Though... testing that the answer to life the universe and everything is equal to the answer to life the universe and everything... isn't very interesting. So next: let's write *real* tests for the `Dinosaur` class.

Chapter 3: Testing Class Methods

As a reminder, the class is currently pretty simple: we pass some data to the constructor..and then we can *read* that data via some methods. Instead of just "hoping" this all works, let's go ahead and make sure that our `Dinosaur` class is *really* bug-free with some tests!

In `DinosaurTest`, remove these two tests and replace them with `public function testCanGetAndSetData()`:

25 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
    public function testCanGetAndSetData(): void
    {
... lines 12 - 22
    }
}
```

Inside... we're literally going to play with the object by instantiating it and trying some methods.

So, `$dino = new Dinosaur()` .... and pass in some data. For the name, eh - let's use `Big Eaty`: he's our resident `Tyrannosaurus` who happens to be `15` meters in length. And Big Eaty is currently living in `Paddock A`:

25 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
    public function testCanGetAndSetData(): void
    {
        $dino = new Dinosaur(
            name: 'Big Eaty',
            genus: 'Tyrannosaurus',
            length: 15,
            enclosure: 'Paddock A',
        );
... lines 18 - 22
    }
}
```

Now that we have our `Dinosaur` object, we can write a few assertions. `self::assertSame()` that `Big Eaty` is identical to `$dino->getName()`, `assertSame()` that `Tyrannosaurus` is identical to `$dino->getGenus()`, `assertSame()` that `15` is identical to `getLength()`, and last but not least, `assertSame()` that Big Eaty is *still* in `Paddock A` when we call `getEnclosure()` ... and not running wild around the island:

25 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
    public function testCanGetAndSetData(): void
    {
        $dino = new Dinosaur(
            name: 'Big Eaty',
            genus: 'Tyrannosaurus',
            length: 15,
            enclosure: 'Paddock A',
        );

        self::assertSame('Big Eaty', $dino->getName());
        self::assertSame('Tyrannosaurus', $dino->getGenus());
        self::assertSame(15, $dino->getLength());
        self::assertSame('Paddock A', $dino->getEnclosure());
    }
}
```

Let's try it! Move back to your terminal and run:

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

Should I Test that Method?

And... YES! We have one test with four assertions. But... looking back at our `Dinosaur` class, we're not really doing a whole heck of a lot in here. We're requiring a few arguments in our constructor, setting them on properties, and exposing those properties with getter methods. Nothing complex at all. So while our `DinosaurTest` is *perfectly* acceptable, it's not the *most* useful. Because the odds of these methods having a bug are low. And besides, if there *were* a bug, we'll probably catch it while testing *other* parts of our app that *call* these.

The point is: while you can do whatever you want, this probably isn't a testthat I would write in a *real* project. My rule of thumb is: if a method scares, it's worth a test. And if you're not sure, it's always safe to add a test.

The Order of the assert() Method Arguments

By the way: the argument *order* for the assert methods is important.

The first argument should always be the *expected* argument - like `Big Eaty` - and the second should be the *actual* value we get - like `$dino->getName()`. This isn't a huge deal for the assertions we're using here...though if you reverse this, the error message will be confusing.

It *is* more important for other assertions, like `assertGreaterThan()` ... which we can use to test that `$dino->getLength()` is greater than `10`.

30 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
    public function testCanGetAndSetData(): void
    {
... lines 12 - 18
        self::assertGreaterThan(
            $dino->getLength(),
            10
        );

        self::assertSame('Big Eaty', $dino->getName());
        self::assertSame('Tyrannosaurus', $dino->getGenus());
        self::assertSame(15, $dino->getLength());
        self::assertSame('Paddock A', $dino->getEnclosure());
    }
}
```

When we try this:

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

Yup! One failure in `DinosaurTest`:

> Failed asserting that 10 is greater than 15.

Whoops! Looking back in our `DinosaurTest`, this test failed because we passed the *actual* value first instead of our *expected* value.

The Assert Message

Before we clean this up, let's pass a 3rd *optional* argument:

> Dino is supposed to be bigger than 10 meters.

31 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
    public function testCanGetAndSetData(): void
    {
... lines 12 - 18
        self::assertGreaterThan(
            $dino->getLength(),
            10,
            message: 'Dino is supposed to be bigger than 10 meters!'
        );

        self::assertSame('Big Eaty', $dino->getName());
        self::assertSame('Tyrannosaurus', $dino->getGenus());
        self::assertSame(15, $dino->getLength());
        self::assertSame('Paddock A', $dino->getEnclosure());
    }
}
```

To see what this does, run the tests again:

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

And... sweet! The test still fails but now we *also* see the message, which can sometimes help us more quickly understand *what* failed and why. Every assert method has this "message" argument and I like to use it when a complex test could use a bit more explanation.

Naming Conventions

I want to circle back to the *name* of our first test method: `testCanGetAndSetData`.

31 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
    public function testCanGetAndSetData(): void
    {
... lines 12 - 28
    }
}
```

In standard PHP, we try to create methods that are descriptive... but not necessarily *super* long... since we'll need to call them in our code. Good examples are `getGenus()` and `getName()` in the `Dinosaur` class. But when it comes to testing, keeping things short is *not* a benefit.

Check it out: I change the name of our test method to `testDinosaur()` ... and then run our tests again.

```
vendor/bin/phpunit
```

PHPUnit tells us that `DinosaurTest::testDinosaur()` failed asserting that 10 is greater than 15. Ok... but *what* are we testing? The method name - `testDinosaur()` - tells us nothing... especially since we're *inside* of a class called `DinosaurTest`! Yea, I get it: we're testing dinosaurs!

The *name* of each test method is *your* chance to describe *exactly* what you're testing, and even sometimes *why*. Change the test name back to `testCanGetAndSetData()`, which does a *much* better job of explaining the *purpose* of this test. Notice that it almost reads like a sentence. That's great! And some people even take this further by including the word "it", like `testItCanGetAndSetData()`. The point is: be descriptive, there's no downside to long test names.

Descriptive Testdox Output

Let me show you one more cool trick with PHPUnit. Move back to the terminal and run our tests again... but *this* time pass a `--testdox` flag:

```
./vendor/bin/phpunit --testdox
```

And... Wooah! The output is different. Most importantly, it turned the method name into a human-readable sentence... which is minor, but cool.

By the way, the `phpunit` executable has a lot more options and arguments available. Run PHPUnit with the `help` flag to see them.

```
./vendor/bin/phpunit --help
```

We'll talk about the most useful of these throughout the tutorial.

Before we keep going, we need to cleanup our test.Remove this `testGreaterThan()` assertion...

25 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
    public function testCanGetAndSetData(): void
    {
        $dino = new Dinosaur(
            name: 'Big Eaty',
            genus: 'Tyrannosaurus',
            length: 15,
            enclosure: 'Paddock A',
        );

        self::assertSame('Big Eaty', $dino->getName());
        self::assertSame('Tyrannosaurus', $dino->getGenus());
        self::assertSame(15, $dino->getLength());
        self::assertSame('Paddock A', $dino->getEnclosure());
    }
}
```

and run our tests again:

```
./vendor/bin/phpunit --testdox
```

And... YES! All of our tests are passing.Coming up next, we're going to get philosophical and take a lookat Test Driven Development or simply - TDD.

Chapter 4: TDD - Test Driven Development

All right. So one of the problems is that when Bob, our park ranger, sees the dinosaur size...he can't remember if these are in meters... or centimeters... which makes a big difference when you step into a cage.

A better way might be to just use words like small, medium, or large.So... let's do that!

What is TDD?

But, to add this feature, we're goingto use a philosophy called Test Driven Development or TDD.TDD is basically a buzzword that describes a 4-step process for writing your tests first.

Step 1: Write a test for the feature.Step 2: Run your test and watch it fail...since we haven't created that feature yet!Step 3: Write as little code as possible to get our test to pass. And Step 4: Now that it's passing, refactor your code if needed to make it more awesome

So, to get the Small, Medium, or Large text,I think we should add a new `getSizeDescription()` method to our `Dinosaur` class. *But*, remember, we're going to do this the TDD way,where Step 1 is to write a *test* for that method... even though it doesn't exist yet. Yes, I know that's weird... but it's kinda awesome!

Step 1: Write a test for the Feature

Add `public function` and let's first test that a dinosaur that'sover 10 meters or greater is large:

32 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
8  class DinosaurTest extends TestCase
9  {
... lines 10 - 24
25     public function testDino10MetersOrGreaterIsLarge(): void
26     {
... lines 27 - 29
30     }
31 }
```

Inside, say `$dino = new Dinosaur()` , give him a name, let's use Big Eaty again,since he's a cool dude, and set his length to 10.

Then, `assertSame()` that `Large` will be identical to `$dino->getSizeDescription()` . For our failure message, let's use `This is supposed to be a Large Dinosaur` .

32 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
8  class DinosaurTest extends TestCase
9  {
... lines 10 - 24
25     public function testDino10MetersOrGreaterIsLarge(): void
26     {
27         $dino = new Dinosaur(name: 'Big Eaty', length: 10);
28
29         self::assertSame('Large', $dino->getSizeDescription(), 'This is supposed to be a large Dinosaur');
30     }
31 }
```

Yes, we're *literally* testing a method that doesn't exist yet.That's TDD.

Step 2: Run the test and watch it fail

Ok, step 1 is done. Step 2 is to run our test and make sure it fails.Open up a terminal and then run `./vendor/bin/phpunit` .

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

And... great! 2 tests, 4 assertions, and 1 error. Our new test failed because, of course, we called an undefined method! We kind of knew this would happen. Hm... Did you notice that our "this is supposed to beat large dinosaur" message isn't showing up here? I'll explain why in just a minute.

Step 3: Write simple code to make it pass

Time for step 3 of TDD: write simple code to make this test pass. This part, taken literally, can get kinda funny. Watch: back in our `Dinosaur` class add a new `public function getSizeDescription()` which will return a `string`. Inside... `return 'Large'`:

45 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
class Dinosaur
{
... lines 7 - 39
    public function getSizeDescription(): string
    {
        return 'Large';
    }
}
```

Yup, that's it! Move back to your terminal and re-run the tests.

```
./vendor/bin/phpunit --testdox
```

And... Awesome - They Pass! Well... of *course* the test passed - we hard coded the result we wanted! But, that's *technically* what TDD says: write the *least* amount of code possible to get your test to pass. If your method is too simple after doing this, it means you're missing more tests - like for small or medium dinosaurs - that would force you to *improve* the method. We'll see that in a minute.

But let's be a *bit* more realistic. Say: `if ($this->length >= 10) {`, then `return 'Large'`:

47 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
class Dinosaur
{
... lines 7 - 39
    public function getSizeDescription(): string
    {
        if ($this->length >= 10) {
            return 'Large';
        }
    }
}
```

Run the tests *one* more time to make sure they're still passing:

```
./vendor/bin/phpunit --testdox
```

And... yes! We're still good to go!

Next, let's finish this method the TDD-way:by writing more tests for the missing features first.Then we'll move onto the final - and most fun step of TDD: Refactoring!

Chapter 5: TDD Part 2: Finish & Refactor

Before we move on to the *last* step in TDD, I think we need to add a couple more size description tests for medium and small dinosaurs.

A few more tests

In our `DinosaurTest::class` copy our `testDino10MetersOrGreaterIsLarge` method and rename it to `testDinoBetween5And9MetersIsMedium()`. Inside, change the `length` of our `$dino` from `10` to `5`, use `Medium` for the expected value, and update the message to `Medium` as well. *Finally*, paste the method again for our small dino test, using the name `testDinoUnder5MetersIsSmall()`. Set the length to `4`, assert that `Small` is identical to `getSizeDescription()` and *also* update the message.

46 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 24
    public function testDino10MetersOrGreaterIsLarge(): void
    {
        $dino = new Dinosaur(name: 'Big Eaty', length: 10);

        self::assertSame('Large', $dino->getSizeDescription(), 'This is supposed to be a large Dinosaur');
    }

    public function testDinoBetween5And9MetersIsMedium(): void
    {
        $dino = new Dinosaur(name: 'Big Eaty', length: 5);

        self::assertSame('Medium', $dino->getSizeDescription(), 'This is supposed to be a medium Dinosaur');
    }

    public function testDinoUnder5MetersIsSmall(): void
    {
        $dino = new Dinosaur(name: 'Big Eaty', length: 4);

        self::assertSame('Small', $dino->getSizeDescription(), 'This is supposed to be a small Dinosaur');
    }
}
```

Back in our terminal, run the tests again:

```
./vendor/bin/phpunit --testdox
```

And... they're failing! But not because our method returns the wrong result. They're failing due to a type error on `getSizeDescription()`:

> The return value must be of type string and none is returned.

Do you remember earlier we ran our large dinosaur test *before* writing the method and we didn't see our "this is supposed to be a large dino" message? Well, we don't see it here either... That's because PHP threw an error... and so the `getSizeDescription()` message explodes *before* PHPUnit can run the `assertSame()` method. It's no big deal and we can still use the stack trace to see exactly where things went wrong.

Alrighty, back to the `Dinosaur` class. Lets fix these tests by adding `if ($this->length)` is less than `5`, `return 'Small'`:

55 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
class Dinosaur
{
... lines 7 - 39
    public function getSizeDescription(): string
    {
        if ($this->length >= 10) {
            return 'Large';
        }

        if ($this->length < 5) {
            return 'Small';
        }
... lines 49 - 52
    }
}
```

And `if ($this->length)` is less than `10`, `return 'Medium'`

55 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
class Dinosaur
{
... lines 7 - 39
    public function getSizeDescription(): string
    {
        if ($this->length >= 10) {
            return 'Large';
        }

        if ($this->length < 5) {
            return 'Small';
        }

        if ($this->length < 10) {
            return 'Medium';
        }
    }
}
```

Back to our terminal, run the test again:

```
./vendor/bin/phpunit --testdox
```

And... alright alright alright... they're passing.

Step 4: Refactoring

So let's move on to the *last* step of TDD... and a fun one! Refactoring our code.

Looking at our `getSizeDescription()` method, I think we can clean this up a bit. And the great news is that, because we've covered our method with tests, if we mess something up during refactoring, the tests will tell us! We get to be reckless! It also means that we didn't really need to worry about writing *perfect* code earlier. We just needed to make our tests pass. NOW we can improve things...

Let's change this middle condition to `if ($this->length)` is greater than or equal to `5`, return `Medium`. We can get rid of this last

conditional altogether and just return `Small`:

53 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
class Dinosaur
{
... lines 7 - 39
    public function getSizeDescription(): string
    {
        if ($this->length >= 10) {
            return 'Large';
        }

        if ($this->length >= 5) {
            return 'Medium';
        }

        return 'Small';
    }
}
```

I like that! To see if we messed up, move back to the terminal and run our tests again.

```
./vendor/bin/phpunit --tesdox
```

And... we've done it! That's TDD - write the test, see the test fail,write simple code to see the test pass, then refactor our code. Rinse and repeat.

TDD Is interesting because, by writing our test first, it forces us to thinkabout *exactly* how a feature should work... Instead of just blindly writing code and seeing what comes out. It also helps us focus on *what* we need to code... Without making things too fancy. Yes, I'm guilty of that too... Get your tests to pass, then refactor...Nothing more is needed.

Use the Size Description in our Controller

And now that we have our fancy new method - built via the powers of TDD let's celebrate by *using* it on the site!

Close up our terminal and move to our template: `templates/main/index.html.twig`. Instead of showing the dino's with `dino.length`, change this to `dino.sizeDescription`. Save it, go back to our browser and...refresh.

51 lines | templates/main/index.html.twig

```
... lines 1 - 3
<div class="container volcano mt-4" style="flex-grow: 1;">
... line 5
    <div class="dino-stats-container mt-2 p-3">
        <table class="table table-striped">
... lines 8 - 15
            <tbody>
            {% for dino in dinos %}
                <tr>
... lines 19 - 20
                    <td>{{ dino.sizeDescription }}</td>
                    <td>{{ dino.enclosure }}</td>
                </tr>
            {% endfor %}
            </tbody>
        </table>
    </div>
</div>
... lines 29 - 51
```

Awesome. We have large, medium, and small for the dinosaur's size instead of a number.No way Bob will accidentally wander into the T-Rex enclosure again!

We've just used TDD to make our app a bit more human-friendly.Coming up next, we'll use some of the TDD principles we've learned here to clean up our tests with PHPUnit's data providers!

Chapter 6: Data Providers

We treat our source code as a first-class citizen. That means, among other things, we avoid duplication. Why not do the same with our tests? Our three tests for the size are...repetitive. They test the same thing just with *slightly* different input and then a different assertion. Is there a way to improve this? Absolutely: thanks to PHPUnit Data Providers.

Refactor our tests

Move to the bottom of `DinosaurTest` and add `public function sizeDescriptionProvider()`. Inside, `yield` an array with `[10, 'Large']`, then `yield [5, 'Medium']`, and finally `yield [4, 'Small']`:

39 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 31
    public function sizeDescriptionProvider()
    {
        yield [10, 'Large'];
        yield [5, 'Medium'];
        yield [4, 'Small'];
    }
}
```

Yield is just a fancy way of returning arrays using PHP's built-in Generator function. As you'll see in a minute, these values - like `10` and `large` will become *arguments* to our test.

Alrighty, up in our test method, add an `int $length` argument and then `string $expectedSize`:

42 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 27
    public function testDino10MetersOrGreaterIsLarge(int $length, string $expectedSize): void
    {
... lines 30 - 32
    }
... lines 34 - 40
}
```

Now instead of Big Eaty's length being `10`, use `$length`. And for our assertion, use `$expectedSize` instead of `Large`:

42 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 27
    public function testDino10MetersOrGreaterIsLarge(int $length, string $expectedSize): void
    {
        $dino = new Dinosaur(name: 'Big Eaty', length: $length);

        self::assertSame($expectedSize, $dino->getSizeDescription(), 'This is supposed to be a large Dinosaur');
    }
... lines 34 - 40
}
```

We do not need the medium and small tests anymore, so we can remove *both* of them.

Ok! Move back to your terminal and run our tests:

```
./vendor/bin/phpunit --testdox
```

Uh oh... Our test is failing because! It says:

> ArgumentCountError - Too few arguments were provided. 0 passed and exactly 2 expected.

Tell our test to use the Data Provider

Oops, we never told our test method to *use* the data provider. Move back into our test and add a DocBlock with `@dataProvider sizeDescriptionProvider`:

42 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 24
    /**
     * @dataProvider sizeDescriptionProvider
     */
    public function testDino10MetersOrGreaterIsLarge(int $length, string $expectedSize): void
    {
        $dino = new Dinosaur(name: 'Big Eaty', length: $length);

        self::assertSame($expectedSize, $dino->getSizeDescription(), 'This is supposed to be a large Dinosaur');
    }
... lines 34 - 40
}
```

When PHPUnit 10 gets released, we'll be able to use a fancy `#[DataProvider]` attribute instead of this annotation.

Back to the terminal! Run the tests again:

```
./vendor/bin/phpunit --testdox
```

And... Yes! Our tests are passing!

Message Keys instead of Arguments

In the output, we see that each test ran with datasets 0, 1, & 2. Those are the arrays from the data provider. We can spruce this up a bit... because it's not going to be very helpful later if PHPUnit tells us that dataset `2` failed. Which one is that?

Move back to our test and, down here after the first `yield` statement, add the message key `'10 Meter Large Dino' =>`. Copy and paste this for our medium dino with `5` instead of `10` and this needs to be `Medium`. Do the same for our small dino with `4` and `Small`:

42 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 34
    public function sizeDescriptionProvider()
    {
        yield '10 Meter Large Dino' => [10, 'Large'];
        yield '5 Meter Medium Dino' => [5, 'Medium'];
        yield '4 Meter Small Dino' => [4, 'Small'];
    }
}
```

Back in our terminal, let's see our tests now:

```
./vendor/bin/phpunit --testdox
```

And... Cool Beans! We now have

> Dino 10 meters or greater is large with 10 Meter Large Dino

This looks a lot better than just seeing data set 0...though we do need to fix one more thing.That test method name doesn't make sense anymore. Change it to `testDinoHasCorrectSizeDescriptionFromLength()` .

And, looking at our assertion, the message argument isn't very useful anymore...so let's remove it.

42 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 34
    public function sizeDescriptionProvider()
    {
        yield '10 Meter Large Dino' => [10, 'Large'];
        yield '5 Meter Medium Dino' => [5, 'Medium'];
        yield '4 Meter Small Dino' => [4, 'Small'];
    }
}
```

Return Types Everywhere!

Finally, although not required... We can use either `array` or `\Generator` as the return type for the data provider.Let's go with `\Generator` - after all, we may need those for the park fences one day...

42 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 34
    public function sizeDescriptionProvider(): \Generator
    {
... lines 37 - 39
    }
}
```

To make sure this didn't break anything, try the tests one more time:

```
./vendor/bin/phpunit --testdox
```

Ummm... Awesome! Green Checks Everywhere!

And there you have it, with a little TLC, our tests are now nice and tidy..Coming up next, let's figure out how we can get our Dino's health status from GitHub and use it in our app...

Chapter 7: Incomplete Tests and Dancing Dino's

Bob just told us he needs to display which dinos are accepting lunch in our app..I mean accepting *visitors*. GenLab has strict protocols in place: park guests can visit with *healthy* dinos... but if they're sick, no visitors allowed. To help display this, we need to store the health status of each dino *and* have an easy way to figure out whether or not this means they're accepting visitors...

Let's skip a test...

Let's start by adding a method - `isAcceptingVisitors()` to `Dinosaur`. But, we'll do this the TDD way by writing the test first.In `DinosaurTest` add `public function testIsAcceptingVisitorsByDefault()`. Inside, `$dino = new Dinosaur()` and let's call him `Dennis`:

49 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
8   class DinosaurTest extends TestCase
9   {
... lines 10 - 41
42      public function testIsAcceptingVisitorsByDefault(): void
43      {
44          $dino = new Dinosaur('Dennis');
... lines 45 - 46
47      }
48  }
```

If we simply instantiate a `Dinosaur` and do nothing else, GenLab policy states that it *is* ok to visit that Dinosaur. So `assertTrue()` that Dennis `isAcceptingVisitors()`:

49 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
8   class DinosaurTest extends TestCase
9   {
... lines 10 - 41
42      public function testIsAcceptingVisitorsByDefault(): void
43      {
44          $dino = new Dinosaur('Dennis');
45
46          self::assertTrue($dino->isAcceptingVisitors());
47      }
48  }
```

Below this test, add another called `testIsNotAcceptingVisitorsIfSick()`. And for now, let's be lazy and just say `$this->markTestIncomplete()`:

58 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
8   class DinosaurTest extends TestCase
9   {
... lines 10 - 48
49      public function testIsNotAcceptingVisitorsIfSick(): void
50      {
... lines 51 - 55
56      }
57  }
```

Ok, let's try the tests:

```
./vendor/bin/phpunit --testdox
```

And... no surprise! Our first new test is failing:

> Call to an undefined method.

But, our *next* test has this weird circle ∅ because we marked the test as *incomplete*. I use this sometimes when I know I need to write a test... I'm just not ready to *do* it quite yet. PHPUnit also has a `markSkipped()` method that can be used to skip tests under certain conditions, like if a test should run on PHP 8.1.

Are they accepting visitors?

Anywho, let's get back to coding, shall we... In our `Dinosaur` class, add a `isAcceptingVisitors()` method that returns a `bool`, and inside we'll return `true`.

58 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
class Dinosaur
{
... lines 7 - 52
    public function isAcceptingVisitors(): bool
    {
        return true;
    }
}
```

Let's see what happens when we run our tests now...

```
./vendor/bin/phpunit --testdox
```

And... Yes! `Is accepting visitors by default` ... is now passing! We still have one *incomplete* test as a reminder, but it's not causing our whole test suite to fail.

Sick Dinos - Stay Away!

Let's finish that now. If we peek at the issues on GitHub - GenLab is using labels to identify the "health" of each dino: "Sick" versus "Healthy". Pretty soon, we're going to *read* these labels and use them in our app. To prep for that, we need a way to store the current *health* on each `Dinosaur`.

Inside the test, remove `markAsIncomplete()` and create a `$dino` named `Bumpy` ... he's a triceratops. Now call `$dino->setHealth('Sick')` and then `assertFalse()` that Bumpy `isAcceptingVisitors()`. He's cranky when he's sick.

58 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 7
class DinosaurTest extends TestCase
{
... lines 10 - 48
    public function testIsNotAcceptingVisitorsIfSick(): void
    {
        $dino = new Dinosaur('Bumpy');

        $dino->setHealth('Sick');

        self::assertFalse($dino->isAcceptingVisitors());
    }
}
```

But, no surprise, PHPStorm is telling us:

> Method setHealth() not found inside Dinosaur

So... let's skip running the test and head straight to `Dinosaur` to add a `setHealth()` method that accepts a `string $health` argument... and returns `void`. Inside, say `$this->health = $health` ... then up top, add a `private string $health` property that defaults to `Healthy`:

64 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
class Dinosaur
{
... lines 7 - 10
    private string $health = 'Healthy';
... lines 12 - 58
    public function setHealth(string $health): void
    {
        $this->health = $health;
    }
}
```

Cool! Now we just need to update `isAcceptingVisitors()` to return `$this->health === $healthy` instead of `true`:

64 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
class Dinosaur
{
... lines 7 - 53
    public function isAcceptingVisitors(): bool
    {
        return $this->health === 'Healthy';
    }
... lines 58 - 62
}
```

Fingers crossed our tests are now passing...

```
./vendor/bin/phpunit --testdox
```

And... Mission Accomplished!

Enums are cool for health labels

Now that the tests are passing, I'm thinking we should refactor the `setHealth()` method to only allow `Sick` or `Healthy` ... and not something like `Dancing` ... Inside `src/`, create a new `Enum/` directory then a new class: `HealthStatus`. For the template, select `Enum` and click `OK`. We need `HealthStatus` to be backed by a `: string` ...

10 lines | src/Enum/HealthStatus.php

```
... lines 1 - 2
namespace App\Enum;

enum HealthStatus: string
{
... lines 7 - 8
}
```

And our first `case HEALTHY` will return `Healthy`, then `case SICK` will return `Sick`.

10 lines | src/Enum/HealthStatus.php

```
... lines 1 - 2
namespace App\Enum;

enum HealthStatus: string
{
    case HEALTHY = 'Healthy';
    case SICK = 'Sick';
}
```

On the `Dinosaur::$health` property, default to `HealthStatus::HEALTHY`. And change the property type to `HealthStatus`. Down in `isAcceptingVisitors()`, return true if `$this->health === HealthStatus::HEALTHY`. Below in `setHealth()`, change the argument type from `string` to `HealthStatus`.

66 lines | src/Entity/Dinosaur.php

```
... lines 1 - 4
use App\Enum\HealthStatus;

class Dinosaur
{
... lines 9 - 12
    private HealthStatus $health = HealthStatus::HEALTHY;
... lines 14 - 55
    public function isAcceptingVisitors(): bool
    {
        return $this->health === HealthStatus::HEALTHY;
    }

    public function setHealth(HealthStatus $health): void
    {
        $this->health = $health;
    }
}
```

The last thing to do is use `HealthStatus::SICK` in our test.

59 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 5
use App\Enum\HealthStatus;
... lines 7 - 8
class DinosaurTest extends TestCase
{
... lines 11 - 49
    public function testIsNotAcceptingVisitorsIfSick(): void
    {
... lines 52 - 53
        $dino->setHealth(HealthStatus::SICK);
... lines 55 - 56
    }
}
```

Let's see if we broke anything!

```
./vendor/bin/phpunit --testdox
```

And... Ya! We *didn't* break anything... I'm only a little surprised.

Show which exhibits are open

To fulfill Bob's wishes, open the `main/index.html.twig` template and add an `Accepting Visitors` heading to the table. In the dino loop, create a new `<td>` and call `dino.acceptingVisitors`. We'll show `Yes` if this is true or `No` if we get false.

53 lines | templates/main/index.html.twig

```
... lines 1 - 3
<div class="container volcano mt-4" style="flex-grow: 1;">
... line 5
    <div class="dino-stats-container mt-2 p-3">
        <table class="table table-striped">
            <thead>
            <tr>
... lines 10 - 13
                <th>Accepting Visitors</th>
            </tr>
            </thead>
            <tbody>
            {% for dino in dinos %}
                <tr>
... lines 20 - 23
                    <td>{{ dino.acceptingVisitors ? 'Yes' : 'No' }}</td>
                </tr>
            {% endfor %}
            </tbody>
        </table>
    </div>
</div>
... lines 31 - 53
```

In the browser, refresh the status page... And... WooHoo! All of our dinos *are* accepting visitors... because we haven't set any as "sick" on our code!

But... We already know from looking at GitHub earlier, that some of our dinos *are* sick. Next: let's use GitHub's API to read the labels from our GitHub repository and set the *real* health on each `Dinosaur` so that our dashboard will update in real-time.

Chapter 8: Create a GitHub Service Test

Now that we can see if a `Dinosaur` is accepting visitors on our dashboard, we need to keep the dashboard updated in real-time by using the health status labels that GenLab has applied to several dino issues on GitHub. To do that we'll create a service that will grab those labels using GitHub's API.

Test for our Service First

To test our new service... which doesn't exist yet, inside of `tests/Unit/` create a new `Service/` directory and then a new class: `GithubServiceTest` ... which will extend `TestCase` :

53 lines | templates/main/index.html.twig

```
... lines 1 - 3
<div class="container volcano mt-4" style="flex-grow: 1;">
... line 5
    <div class="dino-stats-container mt-2 p-3">
        <table class="table table-striped">
            <thead>
            <tr>
... lines 10 - 13
                <th>Accepting Visitors</th>
            </tr>
            </thead>
            <tbody>
            {% for dino in dinos %}
                <tr>
... lines 20 - 23
                    <td>{{ dino.acceptingVisitors ? 'Yes' : 'No' }}</td>
                </tr>
            {% endfor %}
            </tbody>
        </table>
    </div>
</div>
... lines 31 - 53
```

I'm creating this in a `Service/` sub-directory because I'm planning to put the class in the `src/Service/` directory. Add method `testGetHealthReportReturnsCorrectHealthStatusForDino` and inside, `$service = new GithubService()` . Yup, that doesn't exist yet either...

Our service will return a `HealthStatus` enum that's created from the health status label on GitHub, so we'll `assertSame()` that `$expectedStatus` is identical to `$service->getHealthReport()` and then pass `$dinoName` . Yup, we'll be using a data provider for this test... where we accept the *name* of the dino to check for their expected health status.

Let's go create that: `public function dinoNameProvider()` that returns a `\Generator` . Our first dataset for the provider will have the key `Sick Dino` , which returns an array with `HealthStatus::SICK` and `Daisy` for the dino's name... because when we checked GitHub a minute ago, Daisy was sick!

Next up is a `Healthy Dino` with `HealthStatus::HEALTHY` who happens to be the one and only `Maverick` . Up on the test method, add a `@dataProvider` annotation so the test uses `dinoNameProvider` ... and then add `HealthStatus $expectedStatus` and `string $dinoName` arguments.

33 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 2
namespace App\Tests\Unit\Service;

use App\Enum\HealthStatus;
use PHPUnit\Framework\TestCase;

class GithubServiceTest extends TestCase
{
    /**
     * @dataProvider dinoNameProvider
     */
    public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
    {
        $service = new GithubService();

        self::assertSame($expectedStatus, $service->getHealthReport($dinoName));
    }

    public function dinoNameProvider(): \Generator
    {
        yield 'Sick Dino' => [
            HealthStatus::SICK,
            'Daisy',
        ];

        yield 'Healthy Dino' => [
            HealthStatus::HEALTHY,
            'Maverick',
        ];
    }
}
```

Let's do this! Find your terminal and run:

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

And... Yup! Just as we expected, we have two errors because:

> The GithubService class cannot be found

Create the service that will call GitHub

To fix that, find a teammate and ask them nicely to create this class for you! TDD - team-driven-development!

I'm kidding: we got this! Inside of `src/`, create a new `Service/` directory. Then we'll need the new class: `GithubService` and inside, add a method: `getHealthReport()` which takes a `string $dinosaurName` and gives back a `HealthStatus` object.

18 lines | src/Service/GithubService.php

```
... lines 1 - 2
namespace App\Service;

use App\Enum\HealthStatus;

class GithubService
{
    public function getHealthReport(string $dinosaurName): HealthStatus
    {
... lines 11 - 15
    }
}
```

Here's the plan: we'll call GitHub's API to get the list of issues for the `dino-park` repository. Then we'll filter those issues to pick the one that matches `$dinosaurName`. Finally, we'll return `HealthStatus::HEALTHY`, unless the issue has a `Status: Sick` label.

Add the use statement in our test

Before we dive into *writing* that method, jump back into our test and chop off the last couple of letters for `GithubService`. With a little PHPStorm Magic... as soon as I type the letter `i` and hit enter, the use statement is automatically added to the test.Thank you JetBrains!

34 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 5
use App\Service\GithubService;
... lines 7 - 8
class GithubServiceTest extends TestCase
{
... lines 11 - 13
    public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
    {
        $service = new GithubService();
... lines 17 - 18
    }
... lines 20 - 32
}
```

Let's see how the tests are looking:

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

And... Ha! Instead of two failures, we now only have one...

> Sick Dino failed asserting that the two variables reference the same object.

Coming up next, we'll add some logic to our `GithubService` to make this test pass!

Chapter 9: GitHub Service: Implementation

Now that we have an idea of what we need the `GithubService` to do, let's add the logic inside that will fetch the issues from the `dino-park` repository using GitHub's API.

Add the client and make a request

To *make* HTTP requests, at your terminal, install Symfony's HTTP Client with:

```
composer require symfony/http-client
```

Inside of `GithubService`, instantiate an HTTP client with `$client = HttpClient::create()`. To make a request, call `$client->request()`. This needs 2 things. 1st: what HTTP method to use, like `GET` or `POST`. In this case, it should be `GET`. 2nd: the URL, which I'll paste in. This will fetch all of the "issues" from the `dino-park` repository via GitHub's API.

30 lines | src/Service/GithubService.php

```
... lines 1 - 5
use Symfony\Component\HttpClient\HttpClient;

class GithubService
{
    public function getHealthReport(string $dinosaurName): HealthStatus
    {
... lines 12 - 13
        $client = HttpClient::create();

        $response = $client->request(
            method: 'GET',
            url: 'https://api.github.com/repos/SymfonyCasts/dino-park/issues'
        );
... lines 20 - 27
    }
}
```

Parse the HTTP Response

Ok, now what? Looking back at the `dino-park` repo, GitHub will return a JSON response that contains the issues we see here. Each issue has a title with a dino's name and if the issue has a label attached to it, we'll get that back too. So, set `$client->request()` to a new `$response` variable. Then, below, `foreach()` over `$response->toArray()` as an `$issue`. The cool thing about using Symfony's HTTP Client is that we don't have to bother transforming the JSON from GitHub into an array - `toArray()` does that heavy lifting for us. Inside this loop, check if the issue title contains the `$dinosaurName`. So `if (str_contains($issue['title'], $dinosaurName))` then we'll `// Do Something` with that issue.

30 lines | src/Service/GithubService.php

```
... lines 1 - 5
use Symfony\Component\HttpClient\HttpClient;

class GithubService
{
    public function getHealthReport(string $dinosaurName): HealthStatus
    {
... lines 12 - 13
        $client = HttpClient::create();

        $response = $client->request(
            method: 'GET',
            url: 'https://api.github.com/repos/SymfonyCasts/dino-park/issues'
        );

        foreach ($response->toArray() as $issue) {
            if (str_contains($issue['title'], $dinosaurName)) {

            }
        }
... lines 26 - 27
    }
}
```

At this point, we've found the issue for our dinosaur.Woo! Now we need to loop over each label to see if we can find the health status. To help, I'll paste in a private method: you can copy this from the code block on this page.

49 lines | src/Service/GithubService.php

```
... lines 1 - 4
use App\Enum\HealthStatus;
... lines 6 - 7
class GithubService
{
... lines 10 - 29
    private function getDinoStatusFromLabels(array $labels): HealthStatus
    {
        $status = null;

        foreach ($labels as $label) {
            $label = $label['name'];

            // We only care about "Status" labels
            if (!str_starts_with($label, 'Status:')) {
                continue;
            }

            // Remove the "Status:" and whitespace from the label
            $status = trim(substr($label, strlen('Status:')));
        }

        return HealthStatus::tryFrom($status);
    }
}
```

This takes an array of labels...and when it finds one that starts with `Status:`, it returns the correct `HealthStatus` enum based on that label.

Now instead of `// Do Something`, say `$health = $this->getDinoStatusFromLabels()` and pass the labels with `$issue['labels']`.

49 lines | src/Service/GithubService.php

```
... lines 1 - 5
use Symfony\Component\HttpClient\HttpClient;

class GithubService
{
    public function getHealthReport(string $dinosaurName): HealthStatus
    {
... lines 12 - 13
        $client = HttpClient::create();

        $response = $client->request(
            method: 'GET',
            url: 'https://api.github.com/repos/SymfonyCasts/dino-park/issues'
        );

        foreach ($response->toArray() as $issue) {
            if (str_contains($issue['title'], $dinosaurName)) {
                $health = $this->getDinoStatusFromLabels($issue['labels']);
            }
        }
... lines 26 - 27
    }

... lines 30 - 49
```

And now we can return `$health`. But... what if an issue doesn't *have* a health status label? Hmm... at the beginning of this method, set the default `$health` to `HealthStatus::HEALTHY` - because GenLab would *never* forget to put a `Sick` label on a dino that isn't feeling well.

49 lines | src/Service/GithubService.php

```
... lines 1 - 7
class GithubService
{
    public function getHealthReport(string $dinosaurName): HealthStatus
    {
        $health = HealthStatus::HEALTHY;

        $client = HttpClient::create();

        $response = $client->request(
            method: 'GET',
            url: 'https://api.github.com/repos/SymfonyCasts/dino-park/issues'
        );

        foreach ($response->toArray() as $issue) {
            if (str_contains($issue['title'], $dinosaurName)) {
                $health = $this->getDinoStatusFromLabels($issue['labels']);
            }
        }

        return $health;
    }
... lines 29 - 49
```

Hmm... Welp, I think we did it!Let's run our tests to be sure.

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

And... Wow! We have 8 tests, 11 assertions, and they're all passing!Shweeet!

Log all of our requests

One last challenge! To help debugging, I want to log a message each time we make a request to the GitHub API.

No problem! We just need to get the logger service.Add a constructor with `private LoggerInterface $logger` to add an argument and property all at once. Right after we call the `request()` method, add `$this->logger->info()` and pass `Request Dino Issues` for the message and also an array with extra context. How about a `dino` key set to `$dinosaurName` and `responseStatus` to `$response->getStatusCode()` .

59 lines | src/Service/GithubService.php

```
... lines 1 - 5
use Psr\Log\LoggerInterface;
... lines 7 - 8
class GithubService
{
    public function __construct(private LoggerInterface $logger)
    {
    }

    public function getHealthReport(string $dinosaurName): HealthStatus
    {
... lines 17 - 25
        $this->logger->info('Request Dino Issues', [
            'dino' => $dinosaurName,
            'responseStatus' => $response->getStatusCode(),
        ]);
... lines 30 - 37
    }
... lines 39 - 57
}
```

Cool! That *shouldn't* have broken anything in our class,but let's run the tests to be sure:

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

And... Ouch! We *did* break something!

> Too few arguments passed to the constructor in GithubService.0 passed 1 expected.

Of course! When we added the `LoggerInterface` argument to `GithubService` , we never updated our test to pass that in.I'll show you how we can do that next using one of PHPUnit's super abilities: mocking.

Chapter 10: Mocking: Test Doubles

So right now, tests are *failing* because we need to pass a `LoggerInterface` instance to the `GithubService` inside of our test. We *could* just create a logger and pass that in. But... That can get a bit hairy. Instantiating a logger object might be simple... but what if it's not? What if we needed to instantiate an object with 5 required constructor args... and some of those are for *other* objects that are *also* tricky to create. Chaos!

Fortunately, PHPUnit has our back: with super mocking abilities!

A Mock Logger

Inside the `GithubServiceTest` create a `$mockLogger` variable set to `$this->createMock(LoggerInterface::class)`. Pass *this* into the `GithubService` service.

37 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 7
use Psr\Log\LoggerInterface;

class GithubServiceTest extends TestCase
{
... lines 12 - 14
    public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
    {
        $mockLogger = $this->createMock(LoggerInterface::class);

        $service = new GithubService($mockLogger);
... lines 20 - 21
    }
... lines 23 - 35
}
```

Let's see what happens when we run the tests now.

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

And... HA! All of our tests are passing again!

But what is a Mock?

Soo... What is this `createMock()` black magic thing that we're using? `createMock()` allows us to pass in a class or interface and get back a "fake" instance of that class or interface. This object is called a mock.

Now I already ready know what you're about to ask...What happens to the message when we call the `info()` method on the mock `LoggerInterface`?

Welp, a whole lotta nothing... Internally, PHPUnit basically creates a fake class that implements `LoggerInterface` ... except that all of the methods are *empty*. They do nothing and return nothing.

That is unless we *tell* it do something different. More on that soon.

By the way, this mock logger is actually called a *test double*. In fact, we'll run across a few different names for mocks like -test doubles, stubs, and mock objects... All of these names effectively mean the *same* thing: fake objects that stand in for real ones. There *are* some subtle differences between the different namesand we'll clue you in along the way.

We Should Always Mock Services

We still have one minor problem with our test. Anytime we run it, we're calling the *real* GitHub API. This is bad mojo... In a *unit* test, you should *never* use *real* services, like API or database calls. Why? The whole point of a unit test is to test that the code inside `GithubService` works. And, ideally, we would do that *independent* of any other layers of our app because...we simply can't control their behavior. For example, what would happen if GitHub's API is offline for maintenance? Or, tomorrow, GenLab changes `Daisy` from sick to healthy! Right now, *both* of those would cause our tests to fail! But they should *not*! The unit test for `GithubService` should only fail if it contains a bug *in* its code, like it's not parsing the labels correctly.

What's the solution? Mock the `HttpClient`.

Refactoring HttpClient to use DependencyInjection

But... we can't do that as long as we're creating the client *inside* of `GitHubService`. Instead, in the constructor, add a `private HttpClientInterface $httpClient` argument.

57 lines | src/Service/GithubService.php

```
... lines 1 - 6
use Symfony\Contracts\HttpClient\HttpClientInterface;

class GithubService
{
    public function __construct(private HttpClientInterface $httpClient, private LoggerInterface $logger)
    {
    }
... lines 14 - 55
}
```

Then call the `request()` method on `$this->httpClient` instead of `$client`. Since we're *now* using dependency injection, we can remove the static `$client` entire, along with the `use` statement above.

57 lines | src/Service/GithubService.php

```
... lines 1 - 8
class GithubService
{
... lines 11 - 14
    public function getHealthReport(string $dinosaurName): HealthStatus
    {
        $health = HealthStatus::HEALTHY;

        $response = $this->httpClient->request(
            method: 'GET',
            url: 'https://api.github.com/repos/SymfonyCasts/dino-park/issues'
        );
... lines 23 - 35
    }
... lines 37 - 55
}
```

Apart from unit testing, this is just a better way to write your code.

In the test, start by giving the `GithubService` an http client *without* mocking - `HttpClient::create()` - just to make sure everything is working as expected.

38 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 8
use Symfony\Component\HttpClient\HttpClient;

class GithubServiceTest extends TestCase
{
... lines 13 - 15
    public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
    {
... lines 18 - 19
        $service = new GithubService(HttpClient::create(), $mockLogger);
... lines 21 - 22
    }
... lines 24 - 36
}
```

Try the tests:

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

And... cool! We didn't break anything...

Mocking the HttpClient

Now we can mock the `HttpClient`. Below `$mockLogger` add, `$mockClient = $this->createMock()` and pass in `HttpClientInterface::class`. Now pass *this* to our service.

39 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 8
use Symfony\Contracts\HttpClient\HttpClientInterface;

class GithubServiceTest extends TestCase
{
... lines 13 - 15
    public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
    {
        $mockLogger = $this->createMock(LoggerInterface::class);
        $mockHttpClient = $this->createMock(HttpClientInterface::class);

        $service = new GithubService($mockHttpClient, $mockLogger);
... lines 22 - 23
    }
... lines 25 - 37
}
```

Back to the terminal to run our tests:

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

And... Oof! Our `Sick Dino` test

> Failed asserting the two variables are the same

Hmm... For `Sick Dino`, we're expecting a `HealthStatus::SICK` for `Daisy`. In our service, we're calling the `request()` method on our mock, making a log entry, then looping over the array that was returned in our response...HA! That's the problem. Remember:

whenever PHPUnit creates a mock object, it strips out all the logic for each method *within* that mock. Yup, we're looping over nothing!

In this case, we need to *teach* the `HttpClient` mock to return a response that contains a matching issue with a `Status: Sick` label. That would let us assert that our label-parsing logic *is* correct.

How do we do that? It's coming up next!

Chapter 11: Mocking: Stubs

Let's take a quick look back at `GithubService` to see *exactly* what it's doing. First, the constructor requires an `HttpClientInterface` object that we use to call GitHub. In return, we get back a `ResponseInterface` that has an array of issue's for the `dino-park` repository. Next we call the `toArray()` method on the response, and iterate over each issue to see if the title contains the `$dinosaurName`, so we can get its status label.

57 lines | src/Service/GithubService.php

```
... lines 1 - 8
class GithubService
{
... lines 11 - 14
    public function getHealthReport(string $dinosaurName): HealthStatus
    {
... lines 17 - 18
        $response = $this->httpClient->request(
            method: 'GET',
            url: 'https://api.github.com/repos/SymfonyCasts/dino-park/issues'
        );
... lines 23 - 28
        foreach ($response->toArray() as $issue) {
... lines 30 - 32
        }
... lines 34 - 35
    }
... lines 37 - 55
}
```

To get our tests to pass, we need to *teach* our fake `httpClient` that when we call the `request()` method, it should give back a `ResponseInterface` object containing data that *we* control. So... let's do that.

Training the Mock on what to Return

Right after `$mockHttpClient`, say `$mockResponse = $this->createMock()` using `ResponseInterface::class` for the class name. Below on `$mockHttpClient`, call, `->method('request')` which `willReturn($mockResponse)`. This tells our mock client that hey, anytime we call the `request()` method on our mock, you need to return *this* `$mockResponse`.

46 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 9
10 use Symfony\Contracts\HttpClient\ResponseInterface;
11
12 class GithubServiceTest extends TestCase
13 {
... lines 14 - 16
17     public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
18     {
... line 19
20         $mockHttpClient = $this->createMock(HttpClientInterface::class);
21         $mockResponse = $this->createMock(ResponseInterface::class);
22
23         $mockHttpClient
24             ->method('request')
25             ->willReturn($mockResponse)
26         ;
... lines 27 - 30
31     }
... lines 32 - 44
45 }
```

We *could* run our tests now, but they would fail. We taught our mock client *what* it should return when we call the `request()` method. *But*, *now* we need to teach our `$mockResponse` what *it* needs to do when we call the `toArray()` method. So right above, lets teach the `$mockResponse` that when we call, `method('toArray')` and it `willReturn()` an array of issues. Because that's what GitHub returns when we call the API.

51 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 9
10 use Symfony\Contracts\HttpClient\ResponseInterface;
11
12 class GithubServiceTest extends TestCase
13 {
... lines 14 - 16
17     public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
18     {
... line 19
20         $mockHttpClient = $this->createMock(HttpClientInterface::class);
21         $mockResponse = $this->createMock(ResponseInterface::class);
22
23         $mockResponse
24             ->method('toArray')
25             ->willReturn([])
26         ;
27
28         $mockHttpClient
29             ->method('request')
30             ->willReturn($mockResponse)
31         ;
... lines 32 - 35
36     }
... lines 37 - 49
50 }
```

For each issue, GitHub gives us the issue's "title", and among other things, an array of "labels". So let's mimic GitHub and make this array include one issue that has `'title' => 'Daisy'`.

And, for the test, we'll pretend she sprained her ankle so add a `labels` key set to an array, that includes `'name' => 'Status: Sick'`.

Let's also create a healthy dino so we can assert that our parsing checks *that* correctly too. Copy this issue and paste it below. Change `Daisy` to `Maverick` and set his label to `Status: Healthy`.

60 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 9
use Symfony\Contracts\HttpClient\ResponseInterface;

class GithubServiceTest extends TestCase
{
... lines 14 - 16
    public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
    {
... line 19
        $mockHttpClient = $this->createMock(HttpClientInterface::class);
        $mockResponse = $this->createMock(ResponseInterface::class);

        $mockResponse
            ->method('toArray')
            ->willReturn([
                [
                    'title' => 'Daisy',
                    'labels' => [['name' => 'Status: Sick']],
                ],
                [
                    'title' => 'Maverick',
                    'labels' => [['name' => 'Status: Healthy']],
                ],
            ])
        ;

        $mockHttpClient
            ->method('request')
            ->willReturn($mockResponse)
        ;
... lines 41 - 44
    }
... lines 46 - 58
}
```

Perfect! Our assertions are already expecting `Daisy` to be sick and `Maverick` to be healthy. So, if our tests pass, it means that all of our label-parsing logic *is* correct.

Fingers crossed, let's try it:

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

And... Awesome! They *are* passing! And the best part about it, we're no longer calling GitHub's API when we run our tests! Imagine the panic we would cause if we had to lock down the park because our tests failed due to the api being offline...or just someone changing the labels up on GitHub, Ya... I don't want that headache either...

Stubs? Mocks?

Remember when we were talking about the different names for mocks? Welp, both `mockResponse` and `mockHttpClient` are now officially called stubs... That's a fancy way of saying fake objects where we *optionally* take control of the values it returns. That's exactly what we are doing with the `willReturn()` method. Again, the terminology isn't too important, but there you go. These are stubs. And yes, every time I teach this, I need to look up these terms to remember exactly what they mean.

Up next, we're going to turn our *stubs* into full-blown mock objects by also testing the data passed *into* the mock.

Chapter 12: Mocking: Mock Objects

Our tests are passing, the dino's are wandering, and life is great!But... let's think about this for a second. In `GithubService`, when we test `getHealthReport()`, we're able to control the `$response` that we get back from `request()` by using a stub. That's great, but it might also be nice to ensure that the service is only calling GitHub one time*and* that it's using the right HTTP method with the correct URL. Could we do that? Absolutely!

Expect a Method to Be Called

In `GithubServiceTest` where we configure the `$mockHttpClient`, add `->expects()`, and pass `self::once()`.

61 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 11
class GithubServiceTest extends TestCase
{
... lines 14 - 16
    public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
    {
... lines 19 - 36
        $mockHttpClient
            ->expects(self::once())
... lines 39 - 40
        ;
... lines 42 - 45
    }
... lines 47 - 59
}
```

Over in the terminal, run our tests...

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

Expecting Specific Arguments

And... Awesome! We've just added an assertion to our mock clientthat requires the `request` method be called *exactly* once. Let's take it a step further and add `->with()` passing `GET` ... and then I'll paste the URL to the GitHub API.

62 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 11
12  class GithubServiceTest extends TestCase
13  {
... lines 14 - 16
17      public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
18      {
... lines 19 - 36
37          $mockHttpClient
38              ->expects(self::once())
39              ->method('request')
40              ->with('GET', 'https://api.github.com/repos/SymfonyCasts/dino-park')
41              ->willReturn($mockResponse)
42          ;
... lines 43 - 46
47      }
... lines 48 - 60
61  }
```

Try the tests again...

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

And... Huh! We have 2 failures:

> Failed asserting that two strings are equal

Hmm... Ah Ha! My copy and paste skills are a bit weak. I missed `/issue` at the end of the URL. Add that.

62 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 11
12  class GithubServiceTest extends TestCase
13  {
... lines 14 - 16
17      public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
18      {
... lines 19 - 36
37          $mockHttpClient
... lines 38 - 39
40              ->with('GET', 'https://api.github.com/repos/SymfonyCasts/dino-park/issues')
... line 41
42          ;
... lines 43 - 46
47      }
... lines 48 - 60
61  }
```

Let's see if that was the trick:

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

Umm... Yes! We're green all day. But best of all, the tests confirm we're using the correct URL and HTTP method when we call GitHub.

But... What if we actually wanted to call GitHub*more* than just once? Or... we wanted to assert that it was not called at all? PHPUnit has us covered. There are a handful of other methods we can call.For example, change `once()` to `never()`.

And watch what happens now:

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

Hmm... Yup, we have two failures because:

> request() was not expected to be called.

That's really nifty! Change the `expects()` back to `once()` and just to be sure we didn't break anything - run the tests again.

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

And... Awesome!

Carefully Applying Assertions

We *could* call `expects()` on our `$mockResponse` to make sure that `toArray()` is being called exactly once in our service. But, do we really care? If it's not being called at all, our test would certainly fail.And if it's being called twice, no big deal!Using `->expects()` and `->with()` are *great* ways to add extra assertions... when you need them. But no need to micromanage how many times something is called or its arguments if that is *not* so important.

Using GitHubService in our App

Now that `GithubService` is fully tested, we can celebrate by *using* it to drive our dashboard! On `MainController::index()`, add an argument: `GithubService $github` to autowire the new service.

33 lines | src/Controller/MainController.php

```
... lines 1 - 5
use App\Service\GithubService;
... lines 7 - 10
class MainController extends AbstractController
{
    #[Route(path: '/', name: 'main_controller', methods: ['GET'])]
    public function index(GithubService $github): Response
    {
... lines 16 - 30
    }
}
```

Next, right below the `$dinos` array, `foreach()` over `$dinos as $dino` and, inside say `$dino->setHealth()` passing `$github->getHealthReport($dino->getName())`.

33 lines | src/Controller/MainController.php

```
... lines 1 - 5
use App\Service\GithubService;
... lines 7 - 10
class MainController extends AbstractController
{
    #[Route(path: '/', name: 'main_controller', methods: ['GET'])]
    public function index(GithubService $github): Response
    {
... lines 16 - 23
        foreach ($dinos as $dino) {
            $dino->setHealth($github->getHealthReport($dino->getName()));
        }
... lines 27 - 30
    }
}
```

To the browser and refresh...

And... What!

> `getDinoStatusFromLabels()` must be `HealthStatus`, `null` returned

What's going on here? By the way, the fact that our unit test passes but our page fails *can* sometimes happen and in a future tutorial, we'll write a functional test to make sure this page actually loads.

The error isn't very obvious, but I think one of our dino's has a status label that we don't know about. Let's peek back at the issues on GitHub and... HA! "Dennis" is causing problems yet again. Apparently he's a bit hungry...

In our `HealthStatus` enum, we *don't* have a case for `Hungry` status labels. Go figure. Is a hungry dinosaur accepting visitors? I don't know - I guess it depends on if you ask the visitor or the dino. Anyways, `Hungry` is *not* a status we expected. So next, let's throw a clear exception if we run into an unknown status and test for that exception.

Chapter 13: Filtering Out Hungry Dino's

Instead of seeing our dinos on the dashboard, we're seeing a `TypeError` for `GithubService`:

> Return value must be of type `HealthStatus`, `null` returned

That's not doing a great jobof telling us *what* the problem really is. Thanks to the stack trace, it looks like it's being caused by a `Status: Hungry` label. Yup! On GitHub, it looks like Dennis is hungry againafter finishing his daily exercise routine.

Our Enum Is Hungry Too

Looking at `HealthStatus`, we don't have a case for hungry dinos:

10 lines | src/Enum/HealthStatus.php

```
... lines 1 - 2
namespace App\Enum;

enum HealthStatus: string
{
    case HEALTHY = 'Healthy';
    case SICK = 'Sick';
}
```

So add `case HUNGRY` that returns `Hungry` ... then refresh the dashboard.

11 lines | src/Enum/HealthStatus.php

```
... lines 1 - 2
namespace App\Enum;

enum HealthStatus: string
{
    case HEALTHY = 'Healthy';
    case SICK = 'Sick';
    case HUNGRY = 'Hungry';
}
```

And... Ya! No more errors...

But, wait... It says that `Dennis` is *not* accepting visitors. He isn't *sick*, just *hungry*. GenLab said only sick dino's should *not* be on exhibit. Besides, who doesn't want to see what happens to the goat?

Test Hungry Dinos Can Have Visitors

In `DinosaurTest`, we need to assert that hungry dino's *can* have visitors. Hmm... I think we might be able to use `testIsNotAcceptingVisitorsIfSick()` for this. Yup, that's what we'll do. Below, add a `healthStatusProvider()` that returns `\Generator` and for the first dataset `yield 'Sick dino is not accepting visitors'`. In the array say `HealthStatus::SICK`, and `false`. Next, `yield 'Hungry dino is accepting visitors'` with `[HealthStatus::HUNGRY, true]`:

65 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 8
class DinosaurTest extends TestCase
{
... lines 11 - 58
    public function healthStatusProvider(): \Generator
    {
        yield 'Sick dino is not accepting visitors' => [HealthStatus::SICK, false];
        yield 'Hungry dino is accepting visitors' => [HealthStatus::HUNGRY, true];
    }
}
```

Above, add the `@dataProvider` annotation so we can use `healthStatusProvider()`. While we're here, rename the method to `testIsAcceptingVisitorsBasedOnHealthStatus` then add the arguments `HealthStatus $healthStatus` and `bool $expectedVisitorStatus`:

68 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 8
class DinosaurTest extends TestCase
{
... lines 11 - 49
    /**
     * @dataProvider healthStatusProvider
     */
    public function testIsAcceptingVisitorsBasedOnHealthStatus(HealthStatus $healthStatus, bool $expectedVisitorStatus): void
    {
... lines 55 - 59
    }

    public function healthStatusProvider(): \Generator
    {
        yield 'Sick dino is not accepting visitors' => [HealthStatus::SICK, false];
        yield 'Hungry dino is accepting visitors' => [HealthStatus::HUNGRY, true];
    }
}
```

Inside set the health with `$healthStatus` then replace `assertFalse()` with `assertSame($expectedStatus)` is identical to `$dino->isAcceptingVisitors()`:

68 lines | tests/Unit/Entity/DinosaurTest.php

```
... lines 1 - 8
class DinosaurTest extends TestCase
{
... lines 11 - 49
    /**
     * @dataProvider healthStatusProvider
     */
    public function testIsAcceptingVisitorsBasedOnHealthStatus(HealthStatus $healthStatus, bool $expectedVisitorStatus): void
    {
... lines 55 - 56
        $dino->setHealth($healthStatus);

        self::assertSame($expectedVisitorStatus, $dino->isAcceptingVisitors());
    }

    public function healthStatusProvider(): \Generator
    {
        yield 'Sick dino is not accepting visitors' => [HealthStatus::SICK, false];
        yield 'Hungry dino is accepting visitors' => [HealthStatus::HUNGRY, true];
    }
}
```

Phew, that was a lot of work!

Filtering Tests

Let's see if that did the trick. Run:

```
./vendor/bin/phpunit --filter testIsAcceptingVisitorsBasedOnHealthStatus
```

See what I did there? To focus on *just* this test, we can add the `--filter` set to the complete or partial nameof a test class, method, or anything in between. This comes in really handy when you have a large test suiteand only want to run one or a few tests.

Anywho, Hungry dino is not accepting visitors is failing:

> Failed asserting that false is true.

Looking at `Dinosaur::isAcceptingVisitors()`, to account for hungry dino's, we need to return `$this->health` does not equal `HealthStatus::SICK`:

66 lines | src/Entity/Dinosaur.php

```
... lines 1 - 6
class Dinosaur
{
... lines 9 - 55
    public function isAcceptingVisitors(): bool
    {
        return $this->health !== HealthStatus::SICK;
    }
... lines 60 - 64
}
```

Let's see what happens when we run:

```
./vendor/bin/phpunit --filter "Hungry dino is accepting visitors"
```

And... boom! Our hungry dino test is now passing, ha! Yup, we can use data provider keys with the `filter` flag too. But to make sure we didn't stop healthy dino's from having visitors, run:

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

Um... Yes! All dots and no errors. Shweet! We didn't wreck the park. Take a look at the dashboard, refresh, and ya! Dennis is able to eat his lunch with park guests once again. Though, I think we should be proactive and throw a more clear exceptionin case we ever see any future status labels that we don't know about. Let's do that next.

Chapter 14: Testing Exceptional Exceptions

Do you remember when we were seeing this exceptionbecause our app *didn't* understand Maverick's "hungry" status? Welp, we've fixed that, but we still need to take care of one minor detail. Next time GenLab throws us a curve ball, like setting "Status: Antsy" on a dino, `GithubService` should throw a *clear* exception that mentions the label.

Where can we throw an exception?

To do that, we're going to take a break from TDD for just a moment.In `getDinoStatusFromLabels()`, if a label has the "Status:" prefix, we chop that off, set what's left on `$status`, and pass that into `tryFrom()` so we can return a `HealthStatus`. I think this would be a good spot to throw an exception if `tryFrom()` returns `null`.

Cut `HealthStatus::tryFrom($status)` from the return and right above add `$health =` and paste. Then `if (null === $health)` we'll `throw new \RuntimeException()` with the message, `sprintf('%s is an unknown status label!')` passing in `$status`. Below return `$health`.

But, if the issue *doesn't* have a status label, we still need to return a `HealthStatus`. So above, replace `$status` with `$health = HealthStatus::HEALTHY`, because unless GenLab adds a "Status: Sick" label, all of our dinos*are* healthy:

64 lines | src/Service/GithubService.php

```
... lines 1 - 8
class GithubService
{
... lines 11 - 37
    private function getDinoStatusFromLabels(array $labels): HealthStatus
    {
        $health = null;

        foreach ($labels as $label) {
... lines 43 - 49
            // Remove the "Status:" and whitespace from the label
            $status = trim(substr($label, strlen('Status:')));

            $health = HealthStatus::tryFrom($status);

            // Determine if we know about the label - throw an exception if we don't
            if (null === $health) {
                throw new \RuntimeException(sprintf('%s is an unknown status label!', $label));
            }
        }

        return $health ?? HealthStatus::HEALTHY;
    }
}
```

Is the exception thrown?

Now, normally we write tests for *return* values. But you can also write tests to verify that the correct*exception* is thrown. So let's do that in `GithubServiceTest`. Hmm... This first test has a lot of the logic we'll need.Copy that and paste it at the bottom. Change the name to `testExceptionThrownWithUnknownLabel` and remove the arguments. Inside, take out the assertion leaving just the call to `$service->getHealthReport()`. And instead of `$dinoName`, pass in `Maverick`. For `$mockResponse`, remove Daisy from `willReturn()` and change Mavericks label from `Healthy` to `Drowsy`:

90 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 11
12  class GithubServiceTest extends TestCase
13  {
... lines 14 - 61
62      public function testExceptionThrownWithUnknownLabel(): void
63      {
... lines 64 - 67
68          $mockResponse
69              ->method('toArray')
70              ->willReturn([
71                  [
72                      'title' => 'Maverick',
73                      'labels' => [['name' => 'Status: Drowsy']],
74                  ],
75              ])
76          ;
... lines 77 - 86
87          $service->getHealthReport('Maverick');
88      }
89  }
```

Alrighty, lets give this a shot:

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

And... Ouch! `GithubServiceTest` failed because of a:

> RuntimeException: Drowsy is an unknown status label!

This is actually *good* news. It means `GithubService` is doing *exactly* what we want it to do. But, how do we make this test pass?

Right before we call `getHealthReport()`, add `$this->expectException()` passing in `\RuntimeException::class`:

92 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 11
12  class GithubServiceTest extends TestCase
13  {
... lines 14 - 61
62      public function testExceptionThrownWithUnknownLabel(): void
63      {
... lines 64 - 67
68          $mockResponse
69              ->method('toArray')
70              ->willReturn([
71                  [
72                      'title' => 'Maverick',
73                      'labels' => [['name' => 'Status: Drowsy']],
74                  ],
75              ])
76          ;
... lines 77 - 86
87          $this->expectException(\RuntimeException::class);
88
89          $service->getHealthReport('Maverick');
90      }
91  }
```

Try the tests again:

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

Um... awesome sauce! We're green!

Prevent typo's in the exception message

But, hmm... If we manage to dork up our code on accident,a `RuntimeException` *could* be coming from someplace else. To make sure we're testing the *correct* exception, say `$this->expectExceptionMessage('Drowsy is an unknown status label!')` :

93 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 11
12  class GithubServiceTest extends TestCase
13  {
... lines 14 - 61
62      public function testExceptionThrownWithUnknownLabel(): void
63      {
... lines 64 - 67
68          $mockResponse
69              ->method('toArray')
70              ->willReturn([
71                  [
72                      'title' => 'Maverick',
73                      'labels' => [['name' => 'Status: Drowsy']],
74                  ],
75              ])
76          ;
... lines 77 - 86
87          $this->expectException(\RuntimeException::class);
88          $this->expectExceptionMessage('Drowsy is an unknown status label!');
89
90          $service->getHealthReport('Maverick');
91      }
92  }
```

Then run our spell checker again:

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

And... HA! We've added another assertion that is passing and we don't have any typo's in our message.WooHoo!

Test more than the exception message

Along with `expectExceptionMessage()` , PHPUnit has expectations for the exception code,object, and even has the ability to pass a regex to match the message. By the way, all of these `expect` methods are just like the `assert` methods. The big difference is that they must be called *before* the action you're testing rather than after. And just like assertions, if we change the expected message from `Drowsy` to `Sleepy` and run the test:

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

Hmm... Yup! We'll see the test fail because `Drowsy` is not `Sleepy` . Let's change that back in the test...And there you have it!

Dinotopia's gates are now open and Bob is much happier nowthat our app is updated in real-time with GenLab!To celebrate, let's make *our* lives a bit easier by using a touchof HttpClient magic to refactor our test.

Chapter 15: Mocking Symfony's Http Client

Having the ability to mock objects in tests is super awesome, and kind of weirdand complex all at the same time. If we have simple objects, like `Dinosaur`, we should avoid the extra lines of codeand just instantiate a real `Dinosaur` for the test. After all, it's pretty easy to control the behavior of `Dinosaur` just by calling its real methods. No need for the mock weirdness.

But, for more complex objects, like `HttpClient`, using the *real* client... can be a headache. The general rule of thumb is to use *mocks* for *complex* objects like, services... but for *simple* objects, like models or entities, just use the real thing.

Looking back at Symfony's HTTP Client, we *were* able to mock both the client *and* the response to control its behavior. *But*, because needing to do this sort of thing is so common, Symfony's HTTP Client comes with some special classes that can do the mocking *for* us. It comes with two real classes specifically made for *testing*: `MockHttpClient` & `MockResponse`. Using PHPUnit's mock system is fine, but these exist to make our life even easier.

Check it out: instead of *creating* a mock for `$mockResponse`, instantiate a `MockResponse()` passing in `json_encode()` with an array to mimic GitHub's API response. Grab Maverick's issue below and paste that into the array. Since `MockResponse` is already configured, remove the `$mockResponse` bits below.

82 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 8
use Symfony\Component\HttpClient\MockHttpClient;
use Symfony\Component\HttpClient\Response\MockResponse;
... lines 11 - 13
class GithubServiceTest extends TestCase
{
... lines 16 - 63
    public function testExceptionThrownWithUnknownLabel(): void
    {
        $mockResponse = new MockResponse(json_encode([
                [
                    'title' => 'Maverick',
                    'labels' => [['name' => 'Status: Drowsy']],
                ],
            ]));
... lines 72 - 79
    }
}
```

For the client, remove `$mockHttpClient` and below, instantiate a new `MockHttpClient()` passing in `$mockResponse` instead. Since we're not doing anything with `$mockLogger`, cut `createMock()`, remove the variable, and paste that as an argument to `GithubService()`.

82 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 8
use Symfony\Component\HttpClient\MockHttpClient;
use Symfony\Component\HttpClient\Response\MockResponse;
... lines 11 - 13
class GithubServiceTest extends TestCase
{
... lines 16 - 63
    public function testExceptionThrownWithUnknownLabel(): void
    {
        $mockResponse = new MockResponse(json_encode([
                [
                    'title' => 'Maverick',
                    'labels' => [['name' => 'Status: Drowsy']],
                ],
            ]));

        $mockHttpClient = new MockHttpClient($mockResponse);
        $service = new GithubService($mockHttpClient, $this->createMock(LoggerInterface::class));
... lines 75 - 79
    }
}
```

Wow, this is looking better already! But, let's see what happens when we run the tests:

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

And... Woah! All of the tests are passing!

But, the assertion count did go down to "16" because `MockHttpClient` and `MockResponse` do *not* actually perform any assertions, like how many times a method is called.

So... what's the actual benefit to using these mock classes? Why not just create our own via PHPUnit? Ha... Check out this diff of `GithubService`. We removed 11 lines of code by using the "built-in" mocks in just one test! Imagine how many lines of code we could remove by using them in all of our tests. Hm... I think that's exactly what we'll do next.

Chapter 16: Setup and Tearing It Down

Let's continue refactoring our test. In the test method, we create a `MockResponse`, `MockHttpClient`, and instantiate `GitHubService` with a mock `LoggerInterface`. We're doing the same thing in this test above. Didn't Ryan say to DRY out our code in another tutorial? Fine... I suppose we'll listen to him.

Start by adding three `private` properties to our class: a `LoggerInterface $mockLogger`, followed by `MockHttpClient $mockHttpClient` and finally `MockResponse $mockresponse`:

95 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 13
14  class GithubServiceTest extends TestCase
15  {
16      private LoggerInterface $mockLogger;
17      private MockHttpClient $mockHttpClient;
18      private MockResponse $mockResponse;
... lines 19 - 93
94  }
```

At the bottom of the test, create a `private function createGithubService()` that requires `array $responseData` then returns `GithubService`. Inside, say `$this->mockResponse = new MockResponse()` that `json_encode()`'s the `$responseData`:

95 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 13
14  class GithubServiceTest extends TestCase
15  {
16      private LoggerInterface $mockLogger;
17      private MockHttpClient $mockHttpClient;
18      private MockResponse $mockResponse;
... lines 19 - 85
86      private function createGithubService(array $responseData): GithubService
87      {
88          $this->mockResponse = new MockResponse(json_encode($responseData));
... lines 89 - 92
93      }
94  }
```

Since we'll be creating the `MockResponse` *after* we instantiate the `MockHttpClient`, which you'll see in a second, we need to pass our response to the client without using the client's constructor. To do that, we can say `$this->mockHttpClient->setResponseFactory($this->mockResponse)`. Finally return a `new GithubService()` with `$this->mockHttpClient` and `$this->mockLogger`.

95 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 13
class GithubServiceTest extends TestCase
{
    private LoggerInterface $mockLogger;
    private MockHttpClient $mockHttpClient;
    private MockResponse $mockResponse;
... lines 19 - 85
    private function createGithubService(array $responseData): GithubService
    {
        $this->mockResponse = new MockResponse(json_encode($responseData));

        $this->mockHttpClient->setResponseFactory($this->mockResponse);

        return new GithubService($this->mockHttpClient, $this->mockLogger);
    }
}
```

We *could* use a constructor to instantiate our mocks and set them on those properties.But PHPUnit will only instantiate our test class *once*, no matter how many test methods it has.And we want to make sure we have fresh mock objects for*each* test run. How can we do that? At the top, add `protected function setUp()` . Inside, say `$this->mockLogger = $this->createMock(LoggerInterface::class)` then `$this->mockHttpClient = new MockHttpClient()` .

98 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 13
class GithubServiceTest extends TestCase
{
    private LoggerInterface $mockLogger;
    private MockHttpClient $mockHttpClient;
    private MockResponse $mockResponse;

    protected function setUp(): void
    {
        $this->mockLogger = $this->createMock(LoggerInterface::class);
        $this->mockHttpClient = new MockHttpClient();
    }
... lines 25 - 88
    private function createGithubService(array $responseData): GithubService
    {
        $this->mockResponse = new MockResponse(json_encode($responseData));

        $this->mockHttpClient->setResponseFactory($this->mockResponse);

        return new GithubService($this->mockHttpClient, $this->mockLogger);
    }
}
```

Down in the test method, cut the response array,then say `$service = $this->createGithubService()` and paste the array.

98 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 13
class GithubServiceTest extends TestCase
{
    private LoggerInterface $mockLogger;
    private MockHttpClient $mockHttpClient;
    private MockResponse $mockResponse;

    protected function setUp(): void
    {
        $this->mockLogger = $this->createMock(LoggerInterface::class);
        $this->mockHttpClient = new MockHttpClient();
    }
... lines 25 - 73
    public function testExceptionThrownWithUnknownLabel(): void
    {
        $service = $this->createGithubService([
            [
                'title' => 'Maverick',
                'labels' => [['name' => 'Status: Drowsy']],
            ],
        ]);
... lines 82 - 85
        $service->getHealthReport('Maverick');
    }

    private function createGithubService(array $responseData): GithubService
    {
        $this->mockResponse = new MockResponse(json_encode($responseData));

        $this->mockHttpClient->setResponseFactory($this->mockResponse);

        return new GithubService($this->mockHttpClient, $this->mockLogger);
    }
}
```

Let's see how our tests are doing in the terminal...

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

And... Ya! Everything is looking good!

The idea is pretty simple: if your test class has a method called `setUp()`, PHPUnit will call it before *each* test method, which gives us fresh mocks at the start of every test. Need to do something *after* each test? Same thing: create a method called `tearDown()`. This isn't as common... but you might do it if you want to cleanup some filesystem changes that were made during the test. In our case, there's no need.

In addition to `setUp()` and `tearDown()`, PHPUnit also has a few other methods,like `setUpBeforeClass()` and `tearDownAfterClass()`. These are called once per *class*, and we'll get more into thoseas they become relevant in future tutorials. And if you were wondering, these methods are called "Fixture Methods" because they help setup any "fixtures" to get your environmentinto a known state for your test.

Anyhow, let's get back to refactoring. For the first test in this class, cut out the response array, select all of this "dead code", add `$service = $this->createGithubService()` then paste in the array. We can remove the `$service` variable below:

83 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 11
12  class GithubServiceTest extends TestCase
13  {
... lines 14 - 26
27      public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
28      {
29          $service = $this->createGithubService([
30              [
31                  'title' => 'Daisy',
32                  'labels' => [['name' => 'Status: Sick']],
33              ],
34              [
35                  'title' => 'Maverick',
36                  'labels' => [['name' => 'Status: Healthy']],
37              ],
38          ]);
... lines 39 - 43
44      }
... lines 45 - 81
82  }
```

But now we need to figure out how to keep these expectations that we were using on the old `$mockHttpClient`. Being able to test that we only call GitHub *once* with the `GET` HTTP Method and that we're using the right URL, is pretty valuable.

Fortunately, those mock classes have special code *just* for this. Below, `assertSame()` that `1` is identical to `$this->mockHttpClient->getRequestCount()` then `assertSame()` that `GET` is identical to `$this->mockResponse->getRequestMethod()`. Finally, copy and paste the URL into `assertSame()` and call `getRequestUrl()` on `mockResponse`. Remove the old `$mockHttpClient` ... and the `use` statements that we're no longer using up top.

83 lines | tests/Unit/Service/GithubServiceTest.php

```
... lines 1 - 11
12  class GithubServiceTest extends TestCase
13  {
... lines 14 - 26
27      public function testGetHealthReportReturnsCorrectHealthStatusForDino(HealthStatus $expectedStatus, string $dinoName): void
28      {
29          $service = $this->createGithubService([
30              [
31                  'title' => 'Daisy',
32                  'labels' => [['name' => 'Status: Sick']],
33              ],
34              [
35                  'title' => 'Maverick',
36                  'labels' => [['name' => 'Status: Healthy']],
37              ],
38          ]);
39
40          self::assertSame($expectedStatus, $service->getHealthReport($dinoName));
41          self::assertSame(1, $this->mockHttpClient->getRequestsCount());
42          self::assertSame('GET', $this->mockResponse->getRequestMethod());
43          self::assertSame('https://api.github.com/repos/SymfonyCasts/dino-park/issues', $this->mockResponse->getRequestUrl());
44      }
... lines 45 - 81
82  }
```

Alrighty, time to check the fences...

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

And... Wow! Everything is still green!

Welp, there you have it... We've helped Bob improve Dinotopia by adding a few small features to the app. But more importantly, we're able to test that those features are working as we intended. Is there more work to be done? Absolutely! We're going to take our app to the next level by adding a persistence layer to store dinos in the database and learn how to write tests for that too. These tests, where you use *real* database connections or make *real* API calls, instead of mocking, are sometimes called integration tests. That's the topic of the next tutorial in this series.

I hope you enjoyed your time here at the park - and thanks for keeping your arms and legs inside the vehicle at all times. If you have any questions, suggestions, or want to ride with Big Eaty in the Jeep - just leave us a comment. Alright, see you in the next episode!