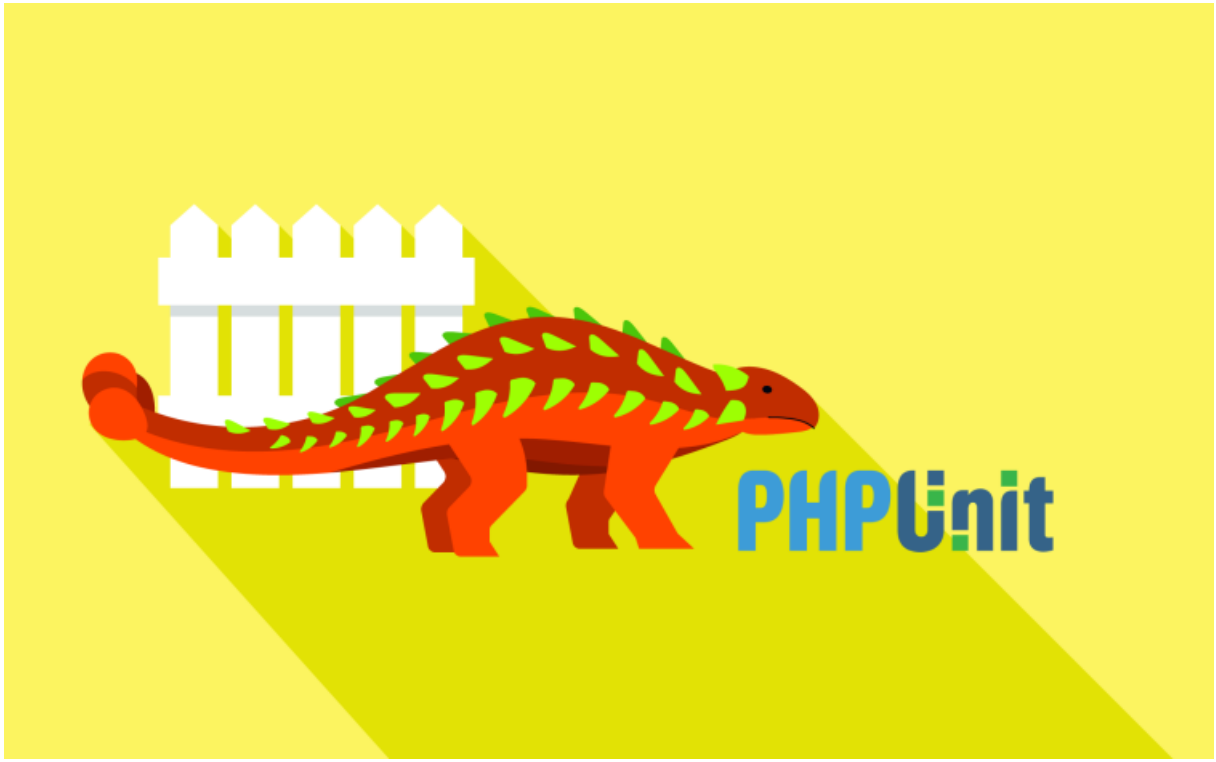
PHPUnit: Integration Testing with Live Services



With <3 from SymfonyCasts

Chapter 1: Hello Integration Tests!

Hey hey, people! Welcome to episode *two* of our testing series, which is all about *integration testing*. In episode 1, Anakin accidentally triggered the auto-pilot on a star fighter... which then taught us all about *unit testing*! What luck!

Unt tests are the *purest* form of testing where you test *classes* and the methods *on* those classes. And if a class requires *other* classes, you *mock* those dependencies. It's cool and *beautiful*... and totally doesn't lead to the dark side, I promise.

In *this* tutorial, things get *messier*, but also more useful in the right situations! Instead of *mocking* dependencies, we're going to test with real *live* services... which sometimes means our tests will cause *real* things to happen, like *actual* queries to the database! That comes with all kinds of exciting complications! And we're going to dive into all of them.

Project Setup

But *first*, let's activate our *own* autopilot and get our app going! Testing is fun, so download the course code from this page and code along with me. After you unzip the file, you'll find a `start/` directory with the same code that you see here, including this nifty `README.md` file. This has all the setup instructions, including database setup, because we *do* have a database in this course. If you were with us for episode one - welcome back - and be sure to download *this* course code because we've changed a few things, like adding a database and upgrading some dependencies.

Oh, and this tutorial uses PHPUnit 9, even though PHPUnit 10 is out. That's fine because there aren't many user-facing changes in PHPUnit 10.

The last step in the README is to find your terminal, move into the project, and run

```
symfony serve -d
```

to start a local web server at https://127.0.0.1:8000. Click that and... here we are! *Dinotopia*: The app where we get to see the status of the dinosaurs inside our park. And *now*, these dinosaurs are coming from the *database*. It's not fancy, but we have a `Dinosaur` entity. And inside our *one* controller, we query for *all* the dinosaurs... and that's what we pass into the template... which is what we see here.

Checking for a "Lock Down"

Everything with the app is working great. Well... except for that one, *minor* problem. You see, sometimes Big Eaty (that's our resident T-Rex) *escapes*, and we don't have a way to lock down the park and notify people. *Basically*, management is worried that *too many* guests are being eaten. So the first feature we're going to build is a system to initiate a lockdown... and we already have an entity for this! It's called, creatively, `LockDown` ... with `$createdAt`, `$endedAt`, and `$status` (which is an `Enum`). Inside the `Enum`, there are three cases: `ACTIVE`, `ENDED`, or `RUN_FOR_YOUR_LIFE`. Let's... try to avoid that last one...

86 lines | src/Entity/LockDown.php

```
... lines 1 - 4
use App\Enum\LockDownStatus;
... lines 6 - 10
class LockDown
{
... lines 13 - 17
    #[ORM\Column]
    private ?\DateTimeImmutable $createdAt = null;

    #[ORM\Column(nullable: true)]
    private ?\DateTimeImmutable $endedAt = null;

    #[ORM\Column(type: Types::STRING, enumType: LockDownStatus::class)]
    private ?LockDownStatus $status = LockDownStatus::ACTIVE;
... lines 26 - 84
}
```

11 lines | src/Enum/LockDownStatus.php

```
... lines 1 - 4
enum LockDownStatus: string
{
    case ACTIVE = 'active';
    case ENDED = 'ended';
    case RUN_FOR_YOUR_LIFE = 'run_for_your_life';
}
```

On our `MainController` (our *homepage*), if the most recent lockdown record in the database has an `ACTIVE` or `RUN_FOR_YOUR_LIFE` status, we need to render a giant warning message on the screen.

39 lines | src/Controller/MainController.php

```
... lines 1 - 12
class MainController extends AbstractController
{
    #[Route(path: '/', name: 'app_homepage', methods: ['GET'])]
    public function index(GithubService $github, DinosaurRepository $repository): Response
    {
        $dinos = $repository->findAll();

        foreach ($dinos as $dino) {
            $dino->setHealth($github->getHealthReport($dino->getName()));
        }

        return $this->render('main/index.html.twig', [
            'dinos' => $dinos,
        ]);
    }
... lines 28 - 37
}
```

To help with this, open `src/Repository/LockDownRepository.php`. To figure out if we're in a lockdown, add a new method called `isInLockDown()` which will return a `bool`. For now, just `return false`.

29 lines | src/Repository/LockDownRepository.php

```
... lines 1 - 16
class LockDownRepository extends ServiceEntityRepository
{
... lines 19 - 23
    public function isInLockDown(): bool
    {
        return false;
    }
}
```

Creating the Test

Let's use some test driven development! Before we write this query, let's add a test for it. We don't have a test for the `LockDownRepository` class yet, so open `tests/`. In the first tutorial, we created a `Unit/` directory and matched the directory structure inside of `src/` for all the classes we need to test.

This time, create a directory called `Integration/`. You don't *need* to organize things like this, but it's fairly common to have unit tests in one directory and integration tests in another. We haven't talked about what an integration test *is* yet, but we'll see that in a minute.

Inside of `Integration/`, we're still going to follow the directory structure. Create a `Repository/` directory since this class lives in `src/Repository/` ... and inside, a new PHP class called `LockDownRepositoryTest`.

Start like we always do: extend `TestCase` from PHPUnit. Call the first method `testIsInLockDownWithNoLockdownRows()`. This will test that, if the lockdown table is empty, then the method should return `false`.

14 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 6
class LockDownRepositoryTest extends TestCase
{
    public function testIsInLockDownWithNoLockDownRows()
    {

    }
}
```

Ok, let's keep pretending that we're living in the world of unit testing and try to test this... like we did in the previous tutorial. To do that, say `$repository = new LockDownRepository()`.

Uh Oh, Instantiating this Object is Hard!

But, hmm. `LockDownRepository` extends `ServiceEntityRepository`, which extends *another* class from Doctrine. If you look, to instantiate it, we need to pass a `ManagerRegistry` from Doctrine. And if you hold "command" or "control" and click into this... and go to the base class, it gets complicated. It calls `$registry->getManagerForClass()` to get the entity manager... and it passes that to the parent. So *already*, we're going to need to mock the registry... so that when `getManagerForClass()` is called, it returns a mocked entity manager.

Inside our repository, we will eventually call `$this->createQueryBuilder()`. If you dive into *that*, it uses the `_em` property (that's that entity manager that we're planning to mock) and calls `createQueryBuilder()`, which returns a `QueryBuilder`. So we also need to mock *this* method on `EntityManager` to return a mock `QueryBuilder`.

This is getting crazy! We have a mock, to return a mock, to return another mock. And ultimately, what would we assert? Would we assert that our code calls the `->andWhere()` method on QueryBuilder with the correct arguments? Or are we going to... somehow have the QueryBuilder generate a *real* query string... then assert that the string... looks correct to us?

Why A Unit Test is the Wrong Tool

No: we're going to do *none* of that. What we're seeing is a situation where a unit test is *not* the right tool. And there are *two* reasons. First, it's too complex! Creating a unit test will require a seemingly never-ending series of mocks. And *second*, a unit test wouldn't be useful! If we're creating a complex query inside of `LockDownRepository`, to make that a *truly* useful test, we need to actually *execute* that query and make sure it returns the results we expect *from* the database.

So, instead of creating a fresh `LockDownRepository` with a bunch of mocks, we're going to ask Symfony to give us the *real* `LockDownRepository`: the one that we would use in our normal code. The one that, when we call a method on it from our test, will execute a *real* query to the database.

That's called an "integration test", and I'll show you how to do it *next*.

Chapter 2: KernelTestCase: Fetching Services

In our app, if we wanted to use `LockDownRepository` to make some real queries, we could autowire `LockDownRepository` into a controller - or somewhere else - call a method on it, and *boom*! Everything would work.

Now we want to do the *same* thing in our test: instead of creating the object *manually*, we want to ask Symfony to give us the *real* service that's configured to talk to the *real* database, so it can do its *real* logic. *Really*!

Booting the Kernel

To fetch a service inside a test, we need to boot up Symfony *then* get access to its *service container*: the mystical object that holds every service in our app.

To help with this, Symfony gives us a base class called `KernelTestCase`. There's nothing particularly special about this class. Hold "command" or "control" to see that it extends the normal `TestCase` from PHPUnit. It just adds methods to boot and shut down Symfony's kernel - that's kind of the heart of Symfony - and to grab the container.

14 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
... line 6
class LockDownRepositoryTest extends KernelTestCase
... lines 8 - 14
```

Fetching Services

At the top of our test method, start with `self::bootKernel()`. Once you call this, you can imagine that you have a Symfony app running in the background, waiting for you to use it. *Specifically*, this means we can grab any *service*. Do that with `$lockDownRepository = self::getContainer()` (which is a helper method from `KernelTestCase`) `->get()`. *Then* pass the service ID which, in our case, is the class name: `LockDownRepository::class`.

To see if this works, `dd($lockDownRepository)`.

18 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 9
    public function testIsInLockDownWithNoLockDownRows()
    {
        self::bootKernel();

        $lockDownRepository = self::getContainer()->get(LockDownRepository::class);
        dd($lockDownRepository);
    }
... lines 17 - 18
```

By the way, unit tests and integration tests generally look the same: you call methods on an object and run assertions. If your test happens to boot the kernel and grab a real service, we give it the name "integration test". But that's just a fancy way of saying: "A unit test... except we use real services".

Okay, let's try this! At your terminal, run:

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

You can also run `./bin/phpunit` - which is a shortcut set up for Symfony. But I'll stick to running `phpunit` directly.

And... yes! *There's* our service! It doesn't look like much, but this lazy object is something that lives in the *real* service.

The Special Test Service Container

So, simple! `self::getContainer` gives us the *service container*... and then we call `get()` on it. But I *do* want to point out that accessing the service container and grabbing a service from it is *not* something we do in our *application* code. For most services, which are private, doing this won't even work! Instead, we rely on dependency injection and *autowiring*.

But in a test, there *is* no dependency injection or autowiring. So, we *need* to grab services like this. And the only reason this even *works* is because `self::getContainer()` gives us a *special* container that *only* exists in the `test` environment. It's *special* because it *does* allow you to call a `get()` method and ask for *any* service you want by its ID... even if that service is normally private. So this is a unique superpower to the `test` environment.

Running Code & Asserting

Ok, since we have `LockDownRepository`, let's try running a simple test. But, hmm, I'm not getting the right autocompletion. Ah, that's because my editor doesn't know *what* the `get()` method returns. To help it, `assert()` that `$lockDownRepository` is an `instanceof LockDownRepository`. This *isn't* a PHPUnit assertion: we didn't say `$this->assert`-something. This is just a PHP function that will throw an exception if `$lockDownRepository` is *not* a `LockDownRepository`. It *will* be... and this code will never cause a problem... but now we enjoy *lovely* autocompletion!

19 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 4
5  use App\Repository\LockDownRepository;
... lines 6 - 7
8  class LockDownRepositoryTest extends KernelTestCase
9  {
10     public function testIsInLockDownReturnsFalseWithNoRows()
11     {
... lines 12 - 14
15         assert($lockDownRepository instanceof LockDownRepository);
... line 16
17     }
18 }
```

Say `$this->assertFalse($lockDownRepository->isInLockDown())`.

19 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 9
10     public function testIsInLockDownReturnsFalseWithNoRows()
11     {
... lines 12 - 15
16         $this->assertFalse($lockDownRepository->isInLockDown());
17     }
... lines 18 - 19
```

The idea is that we haven't added any rows to the database...and *because* of that, we should *not* be in a lockdown. And since the method just returns false right now... this test *should* pass:

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

And... it *does*! So we're using the real service... but it's not, yet, making any queries. Will this keep working if we *do* make a query? Let's find out, *next*.

Chapter 3: Test Environment Database Setup

This first test was *too* easy! So let's write another, *more interesting* one. How about, ahem, `public function testIsInLockDownReturnsTrueIfMostRecentLockdownIsActive()`. Phew!

Start the same as before: `self::bootKernel()`. The *tricky* thing about this test is that we need the database to *not* be empty at the start. We need to insert an active lockdown into the database...so that when we finally call the method and it executes the query, it will find the record.

24 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 7
class LockDownRepositoryTest extends KernelTestCase
{
... lines 10 - 18
    public function testIsInLockDownReturnsTrueIfMostRecentLockDownIsActive()
    {
        self::bootKernel();
    }
}
```

This is a common part of integration tests since they frequently talk to the database.

Seeding the Database

No problem! Let's create a lock down! Add `$lockDown = new LockDown()`, `$lockDown->setReason()` so we know *why* the lockdown is happening, and `$lockDown->setCreatedAt()` to, how about, 1 day ago. That part isn't super important yet. Oh, and we don't need to set the status because, if you look in the class, it defaults to `ACTIVE`.

37 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 20
    public function testIsInLockDownReturnsTrueIfMostRecentLockDownIsActive()
    {
... lines 23 - 24
        $lockDown = new LockDown();
        $lockDown->setReason('Dinos have organized their own lunch break');
        $lockDown->setCreatedAt(new \DateTimeImmutable('-1 day'));
... lines 28 - 34
    }
... lines 36 - 37
```

Saving this is simple too. Grab the `$entityManager` with `self::getContainer()->get(EntityManagerInterface::class)`. And I'll do our `assert()` trick with `$entityManager instanceof EntityManagerInterface` to help my editor. Finish with the usual `$entityManager->persist($lockDown)` and `$entityManager->flush()`.

To see if this is working, down here, `dd($lockDown->getId())`.

37 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 20
    public function testIsInLockDownReturnsTrueIfMostRecentLockDownIsActive()
    {
... lines 23 - 28
        $entityManager = self::getContainer()->get(EntityManagerInterface::class);
        assert($entityManager instanceof EntityManagerInterface);
        $entityManager->persist($lockDown);
        $entityManager->flush();

        dd($lockDown->getId());
    }
... lines 36 - 37
```

Let's try it! Run *just* the tests from this file:

```
./vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

And... *oh*... it *explodes*. Let's see... Ah! It's having trouble connecting to the database!

Forgetting about tests for a moment, this is a familiar problem! The *key* to connecting our app to the database is the `DATABASE_URL` environment variable. I'm using Postgres, but that doesn't matter.

Special .env handling for Tests

Normally, when we set up our local environment, we customize `DATABASE_URL` here in `.env` ... *or* we create a `.env.local` file and override it *there*.

31 lines | .env

```
... lines 1 - 31
```

And, in general, when we boot the kernel in our tests, everything works *exactly* the same as loading our app in the browser. It *does* boot our code in a Symfony environment called `test` instead of `dev` ... and that does change a *few* things. But 99% of the behavior is the same.

If you look at the error, the test is having problems connecting to `127.0.0.1` at port `5432`. That makes sense: it's reading that from our `.env` file. All very normal.

But, there *is* one important difference in the `test` environment. If you create a `.env.local` file, override `DATABASE_URL`, and run your tests (I'll change this port to something crazy like `9999`), it *won't* be used! Check out this error! It's *still* looking for `port 5432`.

In the `test` environment *only*, the `.env.local` file is *not* loaded. So if you want to configure a `DATABASE_URL` *specifically* for your `test` environment, you need to put it into `.env.test`: the environment-specific variable file.

Before we move on, make sure to delete that `.env.local` file to avoid any confusion.

Reading from Docker in your Tests

But in our case, we're *not* going to rely on *any* of these `.env` files. That's because, if you followed the `README.md` instructions, we're using Docker behind the scenes. We have a `docker-compose.yaml` file, which starts a Postgres database. And because we're using the Symfony binary as a web server, it sets the `DATABASE_URL` *automatically* to point to that container.

When we refresh the page... it's *not* using the `DATABASE_URL` from my `.env`: it's using the dynamic value that's set by the `symfony` binary. This is something that we talked about in our Doctrine tutorial.

However, that magic is clearly *not* happening in our test! The error makes it obvious that it's looking at the `DATABASE_URL` from `.env`. And... that's true! This is because the `symfony` binary doesn't have a chance to *inject* the `DATABASE_URL`

environment variable. To *allow* that, instead of running `./vendor/bin/phpunit`, run `symfony php vendor/bin/phpunit` ... followed by the path to the test

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

The `symfony php` command is just a way to execute PHP...but by doing this, it lets the `symfony` binary work its magic.

When we try this... it fails *again*. But check it out! This is a *different* error. Now it's talking about port `58292`. That's the random port that *my* Docker database can apparently be reached on. It also says `database "app_test" does not exist`.

Automatically Suffixed Test Databases

To see what that's about, run:

```
symfony var:export --multiline
```

This shows all the environment variables that the Symfony binary is injecting. The most important is `DATABASE_URL`. This points at the Docker container... which for me, is running on port `58292`.

The key detail is this `app` part. That's the *name* of the database that should be used. So if `DATABASE_URL` is pointing to a database named `app`, why did the error say that a database called `app_test` doesn't exist?

Before we answer that, I have another question: when we run our tests, do we want them to use the *same* database that our local app is using? Ideally, *no*! Having a different database for your tests versus your normal development environment is a good idea. For one... it's just annoying to run your tests and have it manipulate your data while developing. And *fortunately*, having two different databases is something that happens automatically.

Open `config/packages/doctrine.yaml`. Down at the bottom, we have this special `when@test` block. This is config *only* for the `test` environment. And check out that `dbname_suffix`! It's set to `_test`. You can ignore the `%env(default::TEST_TOKEN)%` bit. That relates to a library called ParaTest and, in our case, it will be empty. So *effectively*, it's just `_test`.

49 lines | config/packages/doctrine.yaml

```
... lines 1 - 23
when@test:
    doctrine:
        dbal:
            # "TEST_TOKEN" is typically set by ParaTest
            dbname_suffix: '_test%env(default::TEST_TOKEN)%'
... lines 29 - 49
```

So thanks to this config, in the `test` environment, it takes the `app` config, adds `_test` to it and ultimately uses a database called `app_test`.

That's really nice! And now that we understand that, all *we* need to do is *create* that database.

Creating the Database

At your terminal, run `symfony console` - this is just `bin/console`, but allows the `symfony` binary to inject the `DATABASE_URL` environment variable - `doctrine:database:create --env=test`:

```
symfony console doctrine:database:create --env=test
```

And... success!! We also need to create the schema : doctrine:schema:create

```
symfony console doctrine:schema:create --env=test
```

Cool! Try the test *now*:

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

It worked! That 1 ... comes from the dump down here.

Finishing the Query

Let's finish this test. To make life easier, copy the repository line, then create a new private method: private function getLockDownRepository() . Paste, add return , then the return type. Now we don't need the assert() because PHP will throw a big error if this returns something *else* for some reason.

40 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 9
class LockDownRepositoryTest extends KernelTestCase
{
... lines 12 - 34
    private function getLockDownRepository(): LockDownRepository
    {
        return self::getContainer()->get(LockDownRepository::class);
    }
}
```

Simplify things up here with $this->getLockDownRepository()->isInLockDown() .

40 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 11
    public function testIsInLockDownReturnsFalseWithNoRows()
    {
        self::bootKernel();

        $this->assertFalse($this->getLockDownRepository()->isInLockDown());
    }
... lines 18 - 40
```

Try the test again to make sure it still passes...

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

It *does*. And *interestingly*, the ID is now 2 . More on that soon.

Replace the dump with $this->assertTrue() that $this->getLockDownRepository()->isInLockDown() .

40 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 18
    public function testIsInLockDownReturnsTrueIfMostRecentLockDownIsActive()
    {
... lines 21 - 31
        $this->assertTrue($this->getLockDownRepository()->isInLockDown());
    }
... lines 34 - 40
```

Over in the repository, I'll paste in the real query.This looks for a lockdown that has *not* ended, and returns true or false.

35 lines | src/Repository/LockDownRepository.php

```
... lines 1 - 17
class LockDownRepository extends ServiceEntityRepository
{
... lines 20 - 24
    public function isInLockDown(): bool
    {
        return $this->createQueryBuilder('lock_down')
            ->andWhere('lock_down.status != :endedStatus')
            ->setParameter('endedStatus', LockDownStatus::ENDED)
            ->setMaxResults(1)
            ->getQuery()
            ->getOneOrNullResult() !== null;
    }
}
```

Let's do this!

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

And... the test *fails*? *Oh*, our *second* test passed, but the *original* test is suddenly failing. How did that happen?

It turns out, thanks to the second test, when the *first* test runs, the database is *no longer empty*. In fact, it's piling up with more and more rows each time we run the tests. Watch, run:

```
symfony console dbal:run-sql 'SELECT * FROM lock_down' --env=test
```

Yikes! This is a *critical* problem: we need to guarantee that the database is in a predictable state at the beginning of every test. Let's dive into this very important problem *next*.

Chapter 4: Resetting the Database

It's really common with integration tests or functional tests to talk to the database. And we almost always need to *seed* the database before the test: to add some rows to `LockDown` before doing the work and calling the assertions.

In the first tutorial, we talked about a testing philosophy or pattern called AAA: Arrange, Act, and Assert. With an integration test, the Arrange step commonly involves adding rows to your database, the Act step is where you call the method and then Assert is, of course, the assertions at the end.

Loading Fixtures?

There are two approaches to seeding your database in a test. The first is to write code *inside* the test to insert all the data you need. The second is to create and run a set of fixtures.

And our app *does* have fixtures that power our local site. Should we... load those from inside our test so that it starts with some data in a predictable state?

This sounds nice! But... don't do it! Don't load fixtures in your tests. Why? Because a good test reads like a story: you should be able to read what data is added, what method is called, and what behavior is expected.

If you load a set of fixtures... then suddenly assert that we're in a lockdown, it's not super obvious *why* we're in a lockdown... or what we're even testing! You need to go dig into the app fixtures to find which LockDown records there are... and figure out what's going on. I do *not* like that.

So, even though it might feel like a bit more work, the best strategy is to insert the data you need inside *each* test method. And after the next chapter, it actually *won't* be much work.

Clearing the Data

Even more importantly, no matter how you seed your database, we need to make sure that before each test starts, the database is *empty*. And we just saw why.

Our original test passed... until our second test inserted a row... which caused the first to suddenly fail. Boo. Unless your database is in a *perfectly* predictable state at the start of *each* test, you can't trust them! And the best way to be predictable is to start empty!

We could override the `setUp()` method and run code here that does that. Fortunately, we don't need to because there are *multiple* libraries that already solve this problem. My favorite is Foundry.

Installing zenstruck/foundry

Run:

```
composer require zenstruck/foundry --dev
```

If you watched our Doctrine tutorial, you'll remember Foundry! But you may not know about its testing superpowers... which is where it *really* shines.

The main point of this library is to help create dummy data, and we *are* going to talk about that soon. But it also comes with a super easy way to empty your database between each test.

To use it, at the top of your test class, say use `ResetDatabase` ... and also another trait called `Factories` .

44 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 8
use Zenstruck\Foundry\Test\Factories;
use Zenstruck\Foundry\Test\ResetDatabase;
... line 11
class LockDownRepositoryTest extends KernelTestCase
{
    use ResetDatabase, Factories;
... lines 15 - 42
}
```

Run the tests:

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

They pass! We can run them over and over and over again! Before each individual test method, it empties the database!

By the way, there's another library that does the same thing called `dama/doctrine-test-bundle`, which can be even *faster* than Foundry's `ResetDatabase`. Feel free to install that - then use Foundry just for the factory stuff that we'll talk about soon. They work great together.

Silencing Deprecations with symfony/phpunit-bridge

Before we move on, you probably noticed that we have a bunch of deprecations! Seeing deprecations is great... but an indirect deprecation means that it's not *our* code that's triggering the deprecation: it's one library calling a deprecated method on *another* library.

I'm not too worried about these...so let's silence them for the rest of the tutorial. These deprecation warnings come from Symfony's phpunit-bridge package, and we can *control* how they work.

Open up `phpunit.xml.dist`. Down here, inside the `php` section, add `env` to set an environment variable called `SYMFONY_DEPRECATIONS_HELPER`. For the value, an easy way to silence these warnings is to send them to a log file instead: `logFile=var/log/deprecations.log`.

40 lines | phpunit.xml.dist

```
... lines 1 - 3
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
... lines 5 - 9
>
    <php>
... lines 12 - 17
        <env name="SYMFONY_DEPRECATIONS_HELPER" value="logFile=var/log/deprecations.log"/>
    </php>
... lines 20 - 38
</phpunit>
```

Close that up. Now when we run the tests:

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

Clean and tidy! And the deprecations are still waiting for us in the log file.

Next: let's leverage Foundry Factories to make seeding our database an absolute delight!

Chapter 5: Factory Data Seeding

I have a confession: I've been making us do *way* too much work!

To seed the database, we instantiate the entity, grab the EntityManager, then persist and flush it. There's nothing wrong with this, but Foundry is about to make our life a *lot* easier.

Generating the Factory

At your terminal, run:

```
php bin/console make:factory
```

This command comes from Foundry. I'll select to generate all the factories.

The idea is that you'll create a factory for each entity that you want to create dummy data for, either in a test or for your normal fixtures. We only need `LockDownFactory`, but that's fine.

Spin over and look at `src/Factory/LockDownFactory.php`. I'm not going to talk too much about these factory classes: we already cover them in our Doctrine tutorial. But this class will make it easy to create `LockDown` objects, even setting `createdAt` to a random `DateTime`, `reason` to some random text, and `status` randomly to one of the valid statuses, by default.

72 lines | src/Factory/LockDownFactory.php

```
... lines 1 - 30
31  final class LockDownFactory extends ModelFactory
32  {
... lines 33 - 47
48      protected function getDefaults(): array
49      {
50          return [
51              'createdAt' => \DateTimeImmutable::createFromMutable(self::faker()->dateTime()),
52              'reason' => self::faker()->text(),
53              'status' => self::faker()->randomElement(LockDownStatus::cases()),
54          ];
55      }
... lines 56 - 70
71  }
```

Using the Factory in a Test

Using this in a test is a delight. Say `LockDownFactory::createOne()`. Here, we can pass an array of any field that we want to *explicitly* set. The only thing we care about is that this `LockDown` has an `ACTIVE` status. So, set `status` to `LockDownStatus::ACTIVE`.

41 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 6
use App\Factory\LockDownFactory;
... lines 8 - 13
class LockDownRepositoryTest extends KernelTestCase
{
... lines 16 - 24
    public function testIsInLockDownReturnsTrueIfMostRecentLockDownIsActive()
    {
... lines 27 - 28
        LockDownFactory::createOne([
            'status' => LockDownStatus::ACTIVE,
        ]);
... lines 32 - 33
    }
... lines 35 - 39
}
```

That's it! We don't need to create this `LockDown` and we don't need the EntityManager. That one call takes care of everything.

Watch, when we run the test:

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

It passes! I love that.

Foundry Proxy Objects

By the way, the `LockDownRepository` method returns the new `LockDown` object... which can often be handy. But it's actually wrapped in a special *proxy* object. So if we run the test now, you can see it's a proxy...and the `LockDown` is hiding inside.

Why does Foundry do that? Well, if you go and find their documentation, they have a whole sectionabout using this library inside of tests. One spot talks about the object proxy. The proxy allows you to call all the normal methodson your entity *plus* several additional methods, like `->save()`, `->remove()` or even `->repository()` to get another proxy object that wraps the repository.

So it looks and acts like your normal object, but with some extra methods.That's not important for us right now, I just wanted you to be aware of it. If you do need the *real* entity object, you can call `->object()` to get it.

42 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 24
    public function testIsInLockDownReturnsTrueIfMostRecentLockDownIsActive()
    {
... lines 27 - 31
        dd($lockDown->assertNotPersisted());
... lines 33 - 34
    }
... lines 36 - 42
```

Adding More Objects

Anyway, now that adding data is *so* simple, we can quickly make our test more robust.To see if we can trick my query, call `createMany()` ... to create 5 `LockDown` objects with `LockDownStatus::ENDED`.

To make sure our query looks only at the*newest* `LockDown`, for the active one, set its `createdAt` to `-1 day`. And for the `ENDED`, set these to something older.

46 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 24
25      public function testIsInLockDownReturnsTrueIfMostRecentLockDownIsActive()
26      {
... lines 27 - 28
29          LockDownFactory::createOne([
30              'createdAt' => new \DateTimeImmutable('-1 day'),
31              'status' => LockDownStatus::ACTIVE,
32          ]);
33          LockDownFactory::createMany(5, [
34              'createdAt' => new \DateTimeImmutable('-2 day'),
35              'status' => LockDownStatus::ENDED,
36          ]);
... lines 37 - 38
39      }
... lines 40 - 46
```

Let's see if our query is robust enough to still behave correctly.

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

It is!

But... actually... management has some extra tricky rules around a lockdown. Copy this test, paste it, and rename it to `testIsInLockdownReturnsFalseIfTheMostRecentIsNotActive`.

60 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 40
41      public function testIsInLockDownReturnsFalseIfMostRecentIsNotActive()
42      {
... lines 43 - 52
53      }
... lines 54 - 60
```

To explain management's weird rule, let me tweak the data. Make the first `LockDown` `ENDED` ... then the next, older 5 status `ACTIVE`. Finally, `assertFalse()` at the bottom.

62 lines | tests/Integration/Repository/LockDownRepositoryTest.php

```
... lines 1 - 40
41      public function testIsInLockDownReturnsFalseIfMostRecentIsNotActive()
42      {
43          self::bootKernel();
44
45          LockDownFactory::createOne([
46              'createdAt' => new \DateTimeImmutable('-1 day'),
47              'status' => LockDownStatus::ENDED,
48          ]);
49          LockDownFactory::createMany(5, [
50              'createdAt' => new \DateTimeImmutable('-2 days'),
51              'status' => LockDownStatus::ACTIVE,
52          ]);
53
54          $this->assertFalse($this->getLockDownRepository()->isInLockDown());
55      }
... lines 56 - 62
```

That... might look confusing... and it kind of is. According to management, when determining if we're in lockdown, we should ONLY look at the MOST recent `LockDown` status. If there are older *active* lockdowns... those, apparently, don't matter.

Not surprisingly, when we try the tests:

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

This one *fails*. But, look on the bright side: that test was super-fast to write! And now we can go into `LockDownRepository` to fix things. I'll fast-forward through some changes that fetch the most recent `LockDown` *regardless* of its status.

If we *don't* find *any* lockdowns, return false. Else, I'll add an `assert()` to help my editor... then return true *if* the status does not equal `LockDownStatus::ENDED`.

43 lines | src/Repository/LockDownRepository.php

```
... lines 1 - 17
class LockDownRepository extends ServiceEntityRepository
{
... lines 20 - 24
    public function isInLockDown(): bool
    {
        // find the most recent lock down
        $lockDown = $this->createQueryBuilder('lock_down')
            ->orderBy('lock_down.createdAt', 'DESC')
            ->setMaxResults(1)
            ->getQuery()
            ->getOneOrNullResult();

        if (!$lockDown) {
            return false;
        }

        assert($lockDown instanceof LockDown);

        return $lockDown->getStatus() !== LockDownStatus::ENDED;
    }
}
```

And now:

```
symfony php vendor/bin/phpunit tests/Integration/Repository/LockDownRepositoryTest.php
```

We're green!

Using the LockDown Feature

We've been living in our terminal *so* long that I think we should celebrate by *using* this on our site. In the fixtures, I've added an active `LockDown` by default.

Head over to `MainController` ... and autowire `LockdownRepository $lockdownRepository`. Then throw a new variable in the template called `isLockedDown` set to `$lockdownRepository->isInLockdown()`.

41 lines | src/Controller/MainController.php

```
... lines 1 - 6
use App\Repository\LockDownRepository;
... lines 8 - 13
class MainController extends AbstractController
{
... line 16
    public function index(GithubService $github, DinosaurRepository $repository, LockDownRepository $lockDownRepository): Response
    {
... lines 19 - 24
        return $this->render('main/index.html.twig', [
... line 26
            'isLockedDown' => $lockDownRepository->isInLockDown(),
        ]);
    }
... lines 30 - 39
}
```

Finally, in the template - `templates/main/index.html.twig` - I already have a `_lockdownAlert.html.twig` template. If, `isLockedDown`, include that.

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

```
... lines 1 - 2
{% block body %}
{% if isLockedDown %}
    {{ include('main/_lockDownAlert.html.twig') }}
{% endif %}
... lines 7 - 54
{% endblock %}
```

Moment of truth. Refresh. Run for your life! We are in lockdown!

Next: we need a way to turn a lockdown *off*. Because, if I click this, it... does nothing! To help with this new task, we're going to use an integration test on a different class: on one of our normal services.

Chapter 6: Testing a Service

If you click this button to end the lockdown...it hits a `die` statement. I created a controller... but got lazy...

To end a lockdown, we need to find the active lockdown,change its status to ended, and save it to the database.Easy peasy. But instead of putting that logic inside our controller, let's create a service.

Creating the Service

We *could* use TDD, but I'm going to create the class quickly,and *then* we'll test: it'll be easier to understand.

Inside `src/Service/` , add a new `LockdownHelper` class. I'll paste in some logic... because it's beautifully boring. We have a method called `endCurrentLockDown()` , it calls a `findMostRecent()` method on the repository, sets the status to `ENDED` and flushes. Up here, we autowire `LockdownRepository` and `EntityManagerInterface` .

29 lines | src/Service/LockDownHelper.php

```
... lines 1 - 4
use App\Enum\LockDownStatus;
use App\Repository\LockDownRepository;
use Doctrine\ORM\EntityManagerInterface;

class LockDownHelper
{
    public function __construct(
        private LockDownRepository $lockDownRepository,
        private EntityManagerInterface $entityManager,
    )
    {
    }

    public function endCurrentLockDown(): void
    {
        $lockDown = $this->lockDownRepository->findMostRecent();
        if (!$lockDown) {
            throw new \LogicException('There is no lock down to end');
        }

        $lockDown->setStatus(LockDownStatus::ENDED);
        $this->entityManager->flush();
    }
}
```

The `findMostRecent()` method doesn't exist yet on the repository.So open `LockDownRepository` ... and let's do some refactoring. Create a new public function called `findMostRecent()` , which will return a nullable `Lockdown` . Then grab the code from below, paste, return that and call it: `$lockdown` equals `$this->findMostRecent()` .

45 lines | src/Repository/LockDownRepository.php

```
... lines 1 - 17
class LockDownRepository extends ServiceEntityRepository
{
... lines 20 - 24
    public function findMostRecent(): ?LockDown
    {
        return $this->createQueryBuilder('lock_down')
            ->orderBy('lock_down.createdAt', 'DESC')
            ->setMaxResults(1)
            ->getQuery()
            ->getOneOrNullResult();
    }

    public function isInLockDown(): bool
    {
        $lockDown = $this->findMostRecent();
... lines 37 - 42
    }
}
```

And yes, you could create an integration test for `findMostRecent()`, but we'll skip it.

Back over in `LockDownHelper` ... this is happy! Before we use this class, let's test it!

Unit Test? Or Integration Test?

The first question is, do we need a unit test or an integration test? And honestly, *either* would be fine. We could do a unit test, mock `LockdownRepository`, make sure `findMostRecent()` is called, and that it sets the status to `ENDED` and calls `flush()` on the entity manager. So yea, a unit test *would* be ok: the mocking isn't too complicated... and it *would* test the logic pretty well.

Or we can write an integration test, which will run a bit slower, but be more realistic. For the sake of this tutorial, let's do an integration test. And also, you *could* have both. Heck, there's nothing stopping you from booting the kernel in one test method... and using mocks in another test method in the same class. Mocks and the container are two different tools to help you get your work done.

In the `Integration/` directory, create a new `Service/` directory... then a new PHP class: `LockdownHelperTest`. This time, go straight to extending `KernelTestCase`, then use our two favorite traits: `use ResetDatabaseTrait` and `Factories`.

13 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
use Zenstruck\Foundry\Test\Factories;
use Zenstruck\Foundry\Test\ResetDatabase;

class LockDownHelperTest extends KernelTestCase
{
    use ResetDatabase, Factories;
}
```

Since we'll use these traits in *every* integration test, you can also create a base class. Somewhere inside of `tests/`, you could create an abstract `BaseKernelTestCase`, put the traits there, then have all of your integration tests extend *that*.

Down here, let's whip up our test: `testEndCurrentLockdown()`. And we know how to start: `self::bootKernel()`.

30 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 15
    public function testEndCurrentLockdown()
    {
        self::bootKernel();
... lines 19 - 27
    }
... lines 29 - 30
```

Let's think. If we're going to end a lockdown... we need an active `LockDown` in the database. Say `$lockdown` equals `LockDownFactory::createOne()` ... and pass `status` set to `LockDownStatus::ACTIVE`.

30 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 15
    public function testEndCurrentLockdown()
    {
... lines 18 - 19
        $lockDown = LockDownFactory::createOne([
            'status' => LockDownStatus::ACTIVE,
        ]);
... lines 23 - 27
    }
... lines 29 - 30
```

Since we know our database will start empty, we know *this* will be the item that our query returns. Down here, grab the `$lockDownHelper` with `self::getContainer()->get(LockDownHelper::class)` ... and use the `assert()` trick to tell our editor that this is an `instanceof LockDownHelper`.

30 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 15
    public function testEndCurrentLockdown()
    {
... lines 18 - 23
        $lockDownHelper = self::getContainer()->get(LockDownHelper::class);
        assert($lockDownHelper instanceof LockDownHelper);
... lines 26 - 27
    }
... lines 29 - 30
```

With the "Arrange" part of the test done, let's act: `$lockDownHelper->endCurrentLockDown()`.

With any luck, this record *should* have just changed its status in the database. To prove it, assert that `LockDownStatus::ENDED` equals `$lockDown->getStatus()`.

30 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 15
    public function testEndCurrentLockdown()
    {
... lines 18 - 25
        $lockDownHelper->endCurrentLockDown();
        $this->assertSame(LockDownStatus::ENDED, $lockDown->getStatus());
    }
... lines 29 - 30
```

Auto-Refreshing in Action

That's a good-looking test! Though there is one tiny detail I should mention. First... I'm going to tell a lie. By checking `$lockDown->getStatus()`, we're actually *only* checking that this `LockDown` *object* had its status changed by the code... we're not *actually* testing whether its new value was *saved* to the database. To test that, we could make a fresh query to the database, like via `LockDownFactory::repository()` ... then find the most recent. We'll talk more about the repository shortcut later.

Now, for the truth. You *should* be thinking critically about what you're testing or not testing like we just did. *However*, because we created the `$lockDown` variable through Foundry, it's wrapped in a `Proxy`. One of the *main* features of a `Proxy` is called "auto-refreshing". Each time you access a property or call a method on your entity, behind the scenes, Foundry queries for the *latest* data from the database and sets it. So if we *hadn't* flushed the status change to the database, the test would have *failed*. Foundry actually would have seen that we had *unsaved* changes on that entity, and would have yelled at us. Pretty cool.

Inlined or Removed Services?

Ok, let's try this thing! Run:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

And... what the heck? It says:

> The `LockDownHelper` service or alias has been removed or inlined when the container was compiled.

What does that mean? Ok, a *really* cool thing about Symfony's service container is that if a service isn't used by *anything* in your app, it's *removed* from the container... which makes our app leaner and meaner.

In our actual *application* code, like controllers, repositories & services, nobody is using the `LockDownHelper` service. We're not autowiring it into a controller or a service anywhere. And so, Symfony *removes* this from the container... which means that it's *not* there in the test.

The fix for this is... just to make sure it's used somewhere! I mean, if we're writing this code, certainly we intended to... ya know, *use* it.

In the `endLockDown()` action, autowire `LockDownHelper $lockDownHelper` ... and I'm not even going to call anything on it yet. Just having it here will be enough.

42 lines | src/Controller/MainController.php

```
... lines 1 - 8
use App\Service\LockDownHelper;
... lines 10 - 14
class MainController extends AbstractController
{
... lines 17 - 32
    public function endLockDown(Request $request, LockDownHelper $lockDownHelper)
    {
... lines 35 - 39
    }
}
```

And now:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

The test passes! Woo!

Let's use it: call `$lockDownHelper->endCurrentLockDown()` ... then redirect back to the homepage.

44 lines | src/Controller/MainController.php

```
... lines 1 - 32
    public function endLockDown(Request $request, LockDownHelper $lockDownHelper)
    {
... lines 35 - 38
        $lockDownHelper->endCurrentLockDown();

        return $this->redirectToRoute('app_homepage');
    }
... lines 43 - 44
```

Let's try it! Refresh, we're in a lockdown... "End Lockdown"... it's gone. All the dinos are back in their pens.

Next: I'm going to complicate things by introducing a situation that will make us want to unit test *and* integration test `LockDownHelper` ... at the same time. That'll lead us to something I call "partial mocking".

Chapter 7: Partial Mocking

Let's make `LockDownHelper` more *interesting*. Let's say that, when a lockdown ends, we need to send an API request to GitHub. In our *first* tutorial, we wrote code that made API requests to get info about this `SymfonyCasts/dino-park` repository. Now, we're going to *pretend* that, when we end a lockdown, we need to send an API request to find all the issues with a "lockdown" label and *close* them. We're not... *actually* going to do this, but we'll go through the motions to trigger a fascinating situation.

This Setup: Making API Calls from our Service

In that first tutorial, we made a GitHub service that wraps the API calls. Its one method grabs a health report for the dinosaurs. Add a *new* `public function` called `clearLockDownAlerts()`. Inside, pretend we're making an API call - we don't really need to - then, at least, log a message.

79 lines | src/Service/GithubService.php

```
... lines 1 - 9
class GithubService
{
... lines 12 - 46
    public function clearLockDownAlerts(): void
    {
        $this->logger->info('Cleaning lock down alerts on GitHub...');
        // pretend like this makes an API call to GitHub
    }
... lines 52 - 77
}
```

Cool! Also pretend that we've tested this method in some way - via a unit or integration test. The *point* is: we're *confident* that this method works.

Over in `LockDownHelper`, to make our fake API call, autowire `GithubService $githubService` ... and down here, after `flush()`, say `$this->githubService->clearLockDownAlerts()`.

32 lines | src/Service/LockDownHelper.php

```
... lines 1 - 8
class LockDownHelper
{
    public function __construct(
... lines 12 - 13
        private GithubService $githubService
    )
    {
    }
... line 18
    public function endCurrentLockDown(): void
    {
... lines 21 - 28
        $this->githubService->clearLockDownAlerts();
    }
}
```

Okay! Try the test!

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

We haven't changed anything and... it *still* passes. That makes sense. In our test, we ask Symfony for the `LockDownHelper` and it handles passing the new `GithubService` argument when it *creates* that service. And because `GitHubService` isn't *actually* making a real API call, everything is *fine*.

But what if `GithubService` *did* contain real logic to make an HTTP request to GitHub? *That* could cause a few problems. First, it would definitely slow down our test. Second, it might *fail* because, when it checks the repository, we may not have *any* issues with the `LockDown` label. And *third*, if it *does* find issues with that label, it might close them on our *real* production repository... even though this is just a test.

Furthermore - I know, I'm on a roll - if we wanted to test that the `clearLockDownAlerts()` method *was* actually called, in an integration test, the only way to do that is by making an API call from our test to *seed* the repository with some issues (*creating* an issue with a `LockDown` label), calling the method, then making *another* API request from our test to verify that the issue was closed. *Yikes*. That's *too* much work to check something so simple!

Mocking only Some Services?

I hope you're yelling at your computer:

> Ryan! This is the whole point of mocking - what we learned in the first tutorial!

Yea, totally! If we mocked `GitHubHelper`, we would avoid any API calls *and* have an easy way to assert that the method was called. So, darn, we basically want to mock *one* dependency... but use the *real* services for the *other* dependencies. Is that possible? It is! With something I call "partial mocking".

Injecting a Mock into the Container

When we ask the container for the `LockDownHelper` service, it instantiates the *real* services that it needs and passes them to each of the three arguments. What we *really* want to do is have it pass the *real* service for `$lockDownRepository` and `$entityManager`, but a *mock* for `$githubService`. And Symfony gives us a way to do that!

Check it out. *Before* we ask for `LockDownHelperService`, create a `$githubService` mock set to `$this->createMock(GitHubService::class)`. Below that, say `$githubService->expects()` and, to make sure this fails at first, use `$this->never()` and `->method('clearLockDownAlerts')`.

36 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 12
class LockDownHelperTest extends KernelTestCase
{
... lines 15 - 16
    public function testEndCurrentLockdown()
    {
... lines 19 - 24
        $githubService = $this->createMock(GithubService::class);
        $githubService->expects($this->never())
            ->method('clearLockDownAlerts');
... lines 28 - 33
    }
}
```

If we stop now and run the test:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

It still passes. We created a mock... but no one is *using* it. We need to tell Symfony:

> Hey! Replace the real `GitHubService` in the container with this mock.

Doing that is simple: `self::getContainer()->set()` passing the ID of the service, which is `GithubService::class`, then `$githubService`.

36 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 16
17      public function testEndCurrentLockdown()
18      {
... lines 19 - 27
28          self::getContainer()->set(GithubService::class, $githubService);
... lines 29 - 33
34      }
... lines 35 - 36
```

Suddenly, *that* becomes the service in the container, and *that* is what will be passed to `LockDownHelper` as the *third* argument.

Try the test!

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

Because of the `$this->never()` ... it *fails*! `clearLockDownAlerts()` was *not* expected to be called, but it *was*... since we're calling it down here. That proves the mock was used!

Change the test from `$this->never()` to `$this->once()` and try again...

36 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 16
17      public function testEndCurrentLockdown()
18      {
... lines 19 - 25
26          $githubService->expects($this->once())
... lines 27 - 33
34      }
... lines 35 - 36
```

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

It *passes*! This is *such* a cool strategy.

Next: Let's look at how we can test if our code caused certain *external* things to happen, starting with testing emails.

Chapter 8: The Repository Test Helper

All right, team! We've covered all the *main* parts of integration testing! Woo! And, it's *delightfully* simple: just a *strategy* to grab the *real* services from a container and *test* them, which... *ultimately* gives us a more realistic test.

The *downsides* of integration tests are that they run *slower* than unit tests, and they're often more *complex*... because we need to think about things like clearing and seeding the database. And sometimes, we don't want real things (like API calls) to happen. In this case, we can use a bit of mocking to avoid that. The big takeaway is, like everything, use the right tool - unit testing or integration testing - for the right job. That's *situational* and it's okay to use *both*.

As we near the finish line, let's dive into testing some of the trickier parts in our system: like whether emails were sent or messenger messages were dispatched. To do this, we need to give Bob a new superpower: the ability to put the park into lockdown. Once activated, our app will shoot off an email to the park crew, basically saying:

> Alert! Dinosaurs on the loose!

Creating the Command

Head over to `LockDownHelper`. Down here, create a new method. We'll call this to put the park into lockdown, so how about `public function dinoEscaped()`. Give it a `void` return type and just put some `TODO` comments here outlining what we need to do: save a `LockDown` to the database and send an email.

38 lines | src/Service/LockDownHelper.php

```
... lines 1 - 8
class LockDownHelper
{
... lines 11 - 31
    public function dinoEscaped(): void
    {
        // TODO: create a LockDown & save
        // send an email with subject like "RUUUUUUNNNNNN!!!!"
    }
}
```

To *call* this code and trigger the lockdown, let's create a new console command. At the terminal, run:

```
php bin/console make:command
```

Call it `app:lockdown:start`.

Simple enough! That created a single class in `src/Command/LockdownStartCommand.php`. Inside, autowire a `private LockDownHelper $lockDownHelper` and make sure to call the `parent` constructor.

39 lines | src/Command/LockdownStartCommand.php

```
... lines 1 - 4
use App\Service\LockDownHelper;
... lines 6 - 13
#[AsCommand(
    name: 'app:lockdown:start',
    description: 'Add a short description for your command',
)]
class LockdownStartCommand extends Command
{
    public function __construct(private LockDownHelper $lockDownHelper)
    {
        parent::__construct();
    }
... lines 24 - 37
}
```

Down here, delete pretty much all of this logic...and replace it with `$this->lockDownHelper->dinoEscaped()` and `$io->caution('Lockdown started!!!!!!!!!')`.

39 lines | src/Command/LockdownStartCommand.php

```
... lines 1 - 28
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);

        $this->lockDownHelper->dinoEscaped();
        $io->caution('Lockdown started!!!!!!');

        return Command::SUCCESS;
    }
... lines 38 - 39
```

Dangerous. This method doesn't do anything *yet*, but we can already go ahead and try the command.Copy its name... and run:

```
php bin/console app:lockdown:start
```

Love it!

Creating the Test

Before we get our hands dirty with the new method, let's write a test.But first, let's do that trick where we add a `private function` to help us get the service we're testing: `private function getLockDownHelper()`, which will return a `LockDownHelper`. Inside, copy the code from above... and return it. *Then*, simplify the code up hereto just `$this->getLockDownHelper()->endCurrentLockDown()`.

39 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 12
class LockDownHelperTest extends KernelTestCase
{
... lines 15 - 16
    public function testEndCurrentLockdown()
    {
... lines 19 - 29
        $this->getLockDownHelper()->endCurrentLockDown();
... line 31
    }
... line 33
    private function getLockDownHelper(): LockDownHelper
    {
        return self::getContainer()->get(LockDownHelper::class);
    }
}
```

All right, *now* create the new test method: `public function testDinoEscapedPersistsLockDown()`. Start the same way we always do - by *booting the kernel*. Then call the method with `$this->getLockDownHelper()->dinoEscaped()`.

46 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 33
    public function testDinoEscapedPersistsLockDown()
    {
        self::bootKernel();

        $this->getLockDownHelper()->dinoEscaped();
    }
... lines 40 - 46
```

Cool! It's not interesting, but try the test anyway:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

It doesn't *fail*, but... it *is* risky because we haven't performed any assertions.

Database Assertions via the Repository

What we want to assert is that this *did* insert a row into the database. To do that, we *could* grab the entity manager or our repository service, make a query, and do some assertions using that. *However*, Foundry comes with a nice trick for this.

After we call the method, say `LockDownFactory`. Normally, we would call things like `create` or `createMany`, but this *also* has a method on it named `repository`. This returns an object from Foundry that *wraps* the repository - much like how Foundry wraps our entities in a `Proxy` object. This means we can call *real* repository methods on it - like `findMostRecent()` or `isInLockDown()`. But it *also* has extra stuff, like `assert()`. Say `->assert()->count(1)` to make sure that the there is *one* record in this table. We *could* go further and *fetch* that record to make sure its status is "active", but I'll skip that.

47 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 33
    public function testDinoEscapedPersistsLockDown()
    {
... lines 36 - 38
        LockDownFactory::repository()->assert()->count(1);
    }
... lines 41 - 47
```

Run the test now.

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

This should fail and... it *does*.

I'll go paste in some code that creates the `LockDown` and saves it. Easy peasy *boring* code.

43 lines | src/Service/LockDownHelper.php

```
... lines 1 - 9
class LockDownHelper
... lines 11 - 32
    public function dinoEscaped(): void
    {
        $lockDown = new LockDown();
        $lockDown->setStatus(LockDownStatus::ACTIVE);
        $lockDown->setReason('Dino escaped... NOT good...');
        $this->entityManager->persist($lockDown);
        $this->entityManager->flush();
        // send an email with subject like "RUUUUUUNNNNNN!!!!"
    }
}
```

Try the test now... it passes!

Next: let's send the email and *test* that it was sent. We'll do this with some core Symfony tools and also with *another* library from zenstruck.

Chapter 9: Testing Emails

When we go into lock down, we need to send an email.Before we write the code to do that, let's add an assertion for it.

Asserting an Email is Sent

How? Symfony has our back: it gives us a few methods relatedto emails, like `$this->assertEmailCount()`. We can assert *a lot* of things about emails, but for simplicity's sake, we'll stick to this simple count.

48 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 12
class LockDownHelperTest extends KernelTestCase
{
... lines 15 - 33
    public function testDinoEscapedPersistsLockDown()
    {
... lines 36 - 39
        $this->assertEmailCount(1);
    }
... lines 42 - 46
}
```

Run the test:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

Epic fail, because... we don't even *have* mailer installed yet. Let's do that! Run:

```
composer require symfony/mailer
```

If it asks about Docker configuration, that's up to you,but I'm going to say `Yes permanently`. We'll talk about what that did in a minute, but it's not super important.

Similar to a database, we need to configure our Mailer connection parameters.That's done in `.env` via `MAILER_DSN`. Uncomment this. The `null` transport is a great default. It means that emails *won't* actually be sent in the dev or test environments. And then you can override on your production environment to set it to something real.

35 lines | .env

```
... lines 1 - 32
MAILER_DSN=null://null
... lines 34 - 35
```

If you *do* want to change this to something else in the `dev` environment, I would probably add this `null` transport to `.env.test` ... because it's *really* nice to avoid sending any emails from our tests.

Alright, roll the testing dice again:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

Better! It fails because we haven't sent any emails.Let's do that!

Sending the Email

Over in `LockDownHelper`, autowire one more service: `private MailerInterface $mailer`. Then, down here, since this isn't a *Mailer* tutorial, call a new `sendEmailAlert()` method... and I'll paste that in. Hover over the `Email` class and hit "alt" + "enter"to add the `Symfony\Component\Mime\Email` `use` statement.

58 lines | src/Service/LockDownHelper.php

```
... lines 1 - 8
use Symfony\Component\Mailer\MailerInterface;
use Symfony\Component\Mime\Email;
... line 11
class LockDownHelper
{
    public function __construct(
... lines 15 - 17
        private MailerInterface $mailer
    )
    {
    }
... lines 22 - 35
    public function dinoEscaped(): void
    {
... lines 38 - 42
        $this->sendEmailAlert();
    }

    private function sendEmailAlert(): void
    {
        $email = (new Email())
            ->from('bob@dinotopia.com')
            ->to('staff@dinotopia.com')
            ->subject('PARK LOCKDOWN')
            ->text('RUUUUUUNNNNNN!!!!')
        ;

        $this->mailer->send($email);
    }
}
```

All set! Hustle back to the command-line:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

Got it! The test *passes*!

Seeing Emails via MailCatcher

By the way, this isn't related to testing, but one cool thingsabout using the Docker integration is, when we installed Mailer,it added this `mailcatcher` service.

15 lines | docker-compose.override.yml

```
... lines 1 - 2
services:
... lines 4 - 10
  mailer:
    image: schickling/mailcatcher
    ports: ["1025", "1080"]
... lines 14 - 15
```

Run:

```
docker compose down
```

Then

```
docker compose up -d
```

to start the new service. Then run the test again. It still passes. *However*, because the `mailcatcher` service is running *and* we executed our tests through the Symfony binary, it overrode the `MAILER_DSN` environment variable and *pointed* it at MailCatcher. What... *is* MailCatcher?

To find out, run:

```
symfony open:local:webmail
```

Sweet! MailCatcher is a fake email service with a little web GUIto see the emails your app has sent. If we sent an email via our real app, that would show up here.

Watch. Run:

```
symfony console app:lockdown:start
```

Lockdown! And when you check MailCatcher... ha! We have two messages! Pretty cool!

Using zenstruck/mailer-test

Anyway, before we stop talking about emails, I want to show you one more toolAnd it's *another* library from Zenstruck. Run:

```
composer require zenstruck/mailer-test --dev
```

Symfony has built-in tools for testing emails, and they work *great*. This `mailer-test` library gives us even *more* tools, and it's simple to use!

Add another trait to our test - `use InteractsWithMailer` - and then, down here, instead of `assertEmailCount`, we can say

`$this->mailer()->` ... and then, woh, we have a *ton* of different asserts at our disposal. Say `->assertSentEmailCount(1)`, and below that, `assertEmailSentTo()` with `staff@dinotopia.com` and Subject line `PARK LOCKDOWN`. Whoops! Let me fix my typo. You can see that this is the `expectedTo` and then this is a `callable` where we could assert more things or just pass the expected subject.

51 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 11
use Zenstruck\Mailer\Test\InteractsWithMailer;
... line 13
class LockDownHelperTest extends KernelTestCase
{
... line 16
    use InteractsWithMailer;
... lines 18 - 35
    public function testDinoEscapedPersistsLockDown()
    {
... lines 38 - 41
        $this->mailer()->assertSentEmailCount(1);
        $this->mailer()->assertEmailSentTo('staff@dinotopia.com', 'PARK LOCKDOWN');
    }
... lines 45 - 49
}
```

This is pretty simple, but it's one of the *many* things we can do with this library. Check out the docs to learn about everything.

Run the test again:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

All good! Next up: let's talk about testing *messenger*.

Chapter 10: Testing Messenger

Let's spice up our `LockDownHelper` a bit more, shall we?! When we create a lockdown, instead of sending the email directly, we're going to dispatch a message to *Messenger* and have *it* send the email. Start by installing Messenger:

```
composer require symfony/messenger
```

Lovely! In `.env`, this added a `MESSENGER_TRANSPORT_DSN` which, by default, uses the Doctrine transport type. Though, it won't matter which transport type you use - Doctrine, Redis, whatever. As you'll see, in the `test` environment, we'll override this completely.

42 lines | .env

```
... lines 1 - 39
MESSENGER_TRANSPORT_DSN=doctrine://default?auto_setup=0
... lines 41 - 42
```

Setting up the Test Environment Transport

To make testing *easier*, let's also require *another* package from, you guessed it, Zenstruck!

```
composer require zenstruck/messenger-test --dev
```

Cool! This `messenger-test` library adds a special Messenger transport called `test`. We'll still use Doctrine by default, but now open up `config/packages/messenger.yaml`. Uncomment the `async` transport, which uses `MESSENGER_TRANSPORT_DSN`. Below, under `when@test`, we *override* the `async` transport and set it to the `in-memory` type. Oh, and I need to get rid of one extra space. Perfect!

23 lines | config/packages/messenger.yaml

```
framework:
    messenger:
... lines 3 - 5
        transports:
... line 7
            async: '%env(MESSENGER_TRANSPORT_DSN)%'
... lines 9 - 15
when@test:
    framework:
        messenger:
            transports:
... lines 20 - 21
                async: 'in-memory://'
```

The `in-memory` comes from Symfony and it *is* nice for testing. When it's used, messages are not *really* sent to a transport, but are stored - in memory - on an object during the test...which you can then use to assert that the message is there.

I like that! But the `messenger-test` packages gives us something even better. Change this to `test://`. We'll see what that does in a moment.

23 lines | config/packages/messenger.yaml

```
... lines 1 - 15
when@test:
    framework:
        messenger:
            transports:
... lines 20 - 21
                async: 'test://'
```

Testing that Messages were Dispatched

Before we *dispatch* the message inside our code, head into the test. Here, we want to assert that we sent a message to Messenger. And - surprise, surprise - we're going to use another trait. It's called `InteractsWithMessenger`. Down here, right before we call the method, say `$this->transport()->queue()->assertEmpty()`.

57 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 12
use Zenstruck\Messenger\Test\InteractsWithMessenger;
... line 14
class LockDownHelperTest extends KernelTestCase
{
... lines 17 - 18
    use InteractsWithMessenger;
... lines 20 - 37
    public function testDinoEscapedPersistsLockDown()
    {
... lines 40 - 41
        $this->transport()->queue()->assertEmpty();
... lines 43 - 49
    }
... lines 51 - 55
}
```

Similar to the mailer library, there are *a lot* of different things about messages that we can check. We're asserting that the queue starts *empty*, which isn't *really* necessary - but it's a nice way for us to start. At the end, also `assertCount()` that `1` message was sent.

57 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 37
    public function testDinoEscapedPersistsLockDown()
    {
... lines 40 - 48
        $this->transport()->queue()->assertCount(1);
    }
... lines 51 - 57
```

Let's try this! Keep running all of the tests from `LockDownHelper`:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

And... it fails with the exact message we wanted!

> Expected 1 messages, but 0 messages found.

Creating & Dispatching the Message

Sweet! Generate a Messenger message with:

```
./bin/console make:message
```

Call it `LockDownStartedNotification` and put this into the `async` transport. Done! This created a message class, a *handler* class, and also updated `messenger.yaml` so that this class is sent to the `async` transport.

24 lines | config/packages/messenger.yaml

```
framework:
    messenger:
... lines 3 - 11
        routing:
            App\Message\LockDownLiftedNotification: async
... lines 14 - 24
```

Next, waltz into `LockDownHelper` to *dispatch* that. On top, add a `private MessageBusInterface $messageBus`. Then, at the bottom, say `$this->messageBus->dispatch(new LockDownStartedNotification())`.

48 lines | src/Service/LockDownHelper.php

```
... lines 1 - 10
use Symfony\Component\Messenger\MessageBusInterface;
... lines 12 - 13
class LockDownHelper
{
    public function __construct(
... lines 17 - 20
    )
    {
    }
... lines 24 - 37
    public function dinoEscaped(): void
    {
... lines 40 - 44
        $this->messageBus->dispatch(new LockDownLiftedNotification());
    }
}
```

The *handler* for this class, if we look in `src/MessageHandler/LockDownStartedNotification.php`, isn't doing anything *yet*. But this *should* be enough to get our test to pass.

16 lines | src/MessageHandler/LockDownLiftedNotificationHandler.php

```
... lines 1 - 7
#[AsMessageHandler]
final class LockDownLiftedNotificationHandler
{
    public function __invoke(LockDownLiftedNotification $message)
    {
        // do something with your message
    }
}
```

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

And... whoops! A gremlin sneaked into my code! I added the code inside `endCurrentLockDown()` instead of `dinoEscaped()`. And *that's* why we have tests people. When we try again... got it.

Let's move all the mailing logic *out* of this class. Copy the private method, delete where we call it, the `MailerInterface` ... and even the old `use` statements.

Open the handler, paste the private method there and hit "OK" to re-add those `use` statements. Then say `$this->sendEmailAlert()`.

35 lines | src/MessageHandler/LockDownLiftedNotificationHandler.php

```
... lines 1 - 9
#[AsMessageHandler]
final class LockDownLiftedNotificationHandler
{
    public function __construct(private MailerInterface $mailer)
    {

    }

    public function __invoke(LockDownLiftedNotification $message)
    {
        $this->sendEmailAlert();
    }

    private function sendEmailAlert(): void
    {
        $email = (new Email())
            ->from('bob@dinotopia.com')
            ->to('staff@dinotopia.com')
            ->subject('PARK LOCKDOWN')
            ->text('RUUUUUUNNNNNN!!!!')
        ;

        $this->mailer->send($email);
    }
}
```

Cool! Everything should still work fine... except that the test fails:

> Expected 1 emails to be sent, but 0 emails were sent.

Processing Messages in your Test

Hmm. If this were *production*, when we dispatch this message to the `async` transport, it would *not* send the email immediately. It will be sent to a *queue* and processed *later*. And, the `test` transport we're using *works* a lot like a *true* queue. It *receives* the message, but *doesn't* automatically handle it, which is cool. This means that, over in our test, we *are* dispatching this message... but the email is *never* sent because it's still waiting to be processed.

What you do here is up to you. Maybe you're cool just knowing that the message *was* sent.

Or you might want to be a bit more hands-on and say:

> No way! I want full proof that when this message is handled, it sends an email.

We can do that by telling the `test` transport to *process* its messages. Copy those two `mailer()` lines and delete them. Down here, say `$this->transport()->process()`.

59 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 14
15 class LockDownHelperTest extends KernelTestCase
16 {
... lines 17 - 37
38     public function testDinoEscapedPersistsLockDown()
39     {
... lines 40 - 47
48         $this->transport()->process();
49
50         $this->mailer()->assertSentEmailCount(1);
51         $this->mailer()->assertEmailSentTo('staff@dinotopia.com', 'PARK LOCKDOWN');
52     }
... lines 53 - 57
58 }
```

That's it! That will execute the handler for any messages in its queue.Below *that*, the email *should* be sent.

Try it:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

And... it fails. Another bug! *Why* wasn't it sent? Because I was too quick with my handler: there is no `$this->mailer` property. I'm actually surprised that we didn't get a *bigger* error inside our test.

To fix this, add `public function __construct(private MailerInterface $mailer)`. That looks better! And if we try that *again*... it *passes*.

And we can shorten things! Instead of `assertCount(1)` and `->process()`, we can say `processOrFail()`. This method makes sure that there's at least one message to process, and then processes it.

57 lines | tests/Integration/Service/LockDownHelperTest.php

```
... lines 1 - 37
38     public function testDinoEscapedPersistsLockDown()
39     {
... lines 40 - 46
47         $this->transport()->processOrFail();
... lines 48 - 49
50     }
... lines 51 - 57
```

Double-check the test:

```
symfony php vendor/bin/phpunit tests/Integration/Service/LockDownHelperTest.php
```

Got it!

We did team! Our Dinotopia application is *dangerous* and well-tested, thanks to unit *and* integration tests. In the next tutorial in this series, we'll turn to the *final* type of testing - *functional* testing - where you effectively control a browser, navigate to pages and check what's on them. It's fun and can also be used to check JavaScript behavior.

Alright friends, see ya next time.