Test-Driven Drupal

Automated Testing and Test-Driven Development with Drupal

BlogPageTest.php

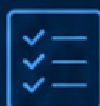
```php
<?php

namespace Drupal\Tests\atdc\Functional;

use Drupal\Tests\BrowserTestBase;
use Symfony\Component\HttpFoundation\Response;

class BlogPageTest extends BrowserTestBase {

  protected $defaultTheme = 'stark';

  public function testBlogPage(): void {
    $this->drupalGet('/blog');

    $this->assertSession()->statusCodeEquals(Response::HTTP_OK);
  }
}
```

RED
Write a failing test

GREEN
Write the minimum code to pass

REFACTOR
Improve the code with confidence

Drupal Native
Test the way Drupal is built



Automated Tests
Reliable tests that run again and again

Better Code
Refactor with confidence and ship quality

Faster Development
Catch regressions early and move faster

OLIVER DAVIES
https://www.oliverdavies.uk/tdd

Test-Driven Drupal

Oliver Davies (opdavies)

Version v0.1.0, 2026-06-09 17:28: Under development

Table of Contents

Zero to Test 2
- Creating a Drupal project 2
- Creating a custom module 2
  - Writing your first test class 2
- Running the test 3
- Configuring PHPUnit 3
- Re-running the tests 4
- Improving the tests 4
- Running the updated tests 5
- Conclusion 6

Diving Deeper 7
- Testing as an authenticated user 7
  - Creating a user 7
- What about content? 8
  - Enabling additional modules 9
- Debugging 9
- Conclusion 9

Building a Blog 10
- Creating the Blog page 10
- Creating a BlogPageController 11
- Asserting posts are visible 12
- Asserting posts are in the correct order 13

Refactoring to a Repository 14
- Creating a PostNodeRepository 14
- Moving the logic 15
- Getting back to green 15
- Creating a service 16
- Injecting more dependencies 16
- Conclusion 17

Testing Post Ordering 18
- Introducing Kernel tests 18
- Writing your first Kernel test 18
  - Resolving setup test failures 19
- Creating posts 20
- Adding assertions for the order 20
- Fixing the ordering 21
- Conclusion 23

Builders and custom assertions 24

- Creating a PostBuilder class . . . 24
- Creating a custom assertion . . . 26

Filling in old tests . . . 28
- Only returning published nodes . . . 28
- Updating PostBuilder . . . 30
- Updating PostNodeRepository . . . 31
- Only returning posts . . . 31
- Conclusion . . . 32

Tagging posts and test configuration . . . 33
- Testing the PostBuilder . . . 33
- Tagging posts . . . 34
- Creating a test module . . . 34
- Creating configuration . . . 35
- Fixing setup issues . . . 37
- Setting tags . . . 38
- Base table or view not found . . . 39
- Adding tag assertions . . . 40
- Conclusion . . . 40

Introducing Unit Tests . . . 41
- Your first Unit test . . . 41
- Wrapping posts . . . 42
- Not wrapping a page . . . 43
- Conclusion . . . 44

Mocking services . . . 45
- Creating the test . . . 45
- Adding the first mock . . . 46
- Getting the posts . . . 46
- Creating nodes and adding assertions . . . 47
- Conclusion . . . 48

Questions and feedback . . . 50

Zero to Test

In this chapter, we start from scratch and end with a working test suite.

Creating a Drupal project

If you don't have one, you'll need a new Drupal project to work on.

You'll need PHP and Composer. For Drupal 11, you'll need PHP 8.3.0 or newer.

First, run `composer create-project drupal/recommended-project atdc-project` followed by `cd atdc-project && composer require --dev drupal/core-dev` to add the development dependencies, including PHPUnit.

At this point, you should have a `web` directory and a `phpunit` file within `vendor/bin`.

Finally, run `php -S localhost:8000 -t web` to start a local web server.

You don't need to install Drupal - as long as you see the installation page, that's fine.

Creating a custom module

Before adding tests, you must create a module to place them in.

Run `mkdir -p web/modules/custom/atdc` to create an empty module directory and create an `atdc.info.yml` file within it with this content:

```
name: ATDC
type: module
core_version_requirement: ^11
package: Custom
```

To make your module compatible with Drupal 10, add `^10` instead of `^11`.

This is the minimum content needed for a module to be installable.

Writing your first test class

Test classes are located in each module's `tests/src` directory.

Run `mkdir -p web/modules/custom/atdc/tests/src/Functional && touch web/modules/custom/atdc/tests/src/Functional/ExampleTest.php` to create the directory structure and a blank test class.

Then, add this content:

```
<?php
```

```
namespace Drupal\Tests\atdc\Functional;

use Drupal\Tests\BrowserTestBase;
use Symfony\Component\HttpFoundation\Response;

class ExampleTest extends BrowserTestBase {

  protected $defaultTheme = 'stark';

}
```

Within a test class, the namespace is `Drupal\Tests\{module_name}` instead of `Drupal\{module_name}`.

With the boilerplate class added, create a test method within it:

```
public function testBasic(): void {
  self::assertTrue(FALSE);
}
```

The class name must be suffixed with `Test`, and the test method must be prefixed with `test` for them to be run.

Now, we have a test with an assertion, and we need to run it and see if it passes.

Running the test

On the command line, run `vendor/bin/phpunit web/modules/custom`, and you'll get an error like:

```
PHPUnit\TextUI\RuntimeException: Class "Drupal\Tests\BrowserTestBase" not found.
```

This isn't an assertion failure, but PHPUnit needs help finding the files it needs to run.

To fix this, let's configure PHPUnit.

Configuring PHPUnit

Create a new `phpunit.xml.dist` file at the root of your project with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="web/core/tests/bootstrap.php" colors="true">
  <php>
    <env name="SIMPLETEST_BASE_URL" value="http://localhost:8000"/>
    <env name="SIMPLETEST_DB" value="sqlite://localhost//dev/shm/test.sqlite"/>
    <ini name="error_reporting" value="32767"/>
```

```
    <ini name="memory_limit" value="-1"/>
  </php>
  <testsuites>
    <testsuite name="Example tests">
      <directory suffix="Test.php">./web/modules/**</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

This is based on `web/core/phpunit.xml.dist` with project-specific changes.

Namely, setting the `bootstrap` value to include the `web/core` path, fixing the error, and populating the `SIMPLETEST_BASE_URL` and `SIMPLETEST_DB` environment variables.

PHPUnit now knows where the files are, to connect to Drupal at http://localhost:8000 (matching the PHP web server address) and an SQLite database.

Drupal 11 requires a newer version of SQLite than Drupal 10. If you get an SQLite error, you may need to update it, or use MySQL by setting your database to something like `mysql://user:password@localhost/databasename#test`, but this means you will need a MySQL server installed and running.

I've also added a `testsuite` that declares where any test classes will be located so the path doesn't need to be specified on the command line.

Re-running the tests

Re-running the tests will give the expected error about a failing assertion:

> Failed asserting that false is true.

Fix the assertion in the test by changing `FALSE` to `TRUE`, run `vendor/bin/phpunit` again, and you should see a passing test.

```
OK (1 test, 2 assertions)
```

Improving the tests

Now you have as passing test and know PHPUnit is working, let's improve it.

Instead of the basic check, let's check whether certain pages exist and are accessible.

To keep things simple and focused on writing and running tests, let's use some standard Drupal pages - the front and administration pages instead of writing your own.

As you're writing functional tests by extending `BrowserTestBase`, you can make HTTP requests to the web server and make assertions on the responses.

Replace the `testBasic` test method with the following:

```
public function testFrontPage(): void {
  $this->drupalGet('/');

  $this->assertSession()->statusCodeEquals(Response::HTTP_FORBIDDEN);
}

public function testAdminPage(): void {
  $this->drupalGet('/admin');

  $this->assertSession()->statusCodeEquals(Response::HTTP_OK);
}
```

These tests will make HTTP requests to the specified paths and assert the status code on the response matches the expected values.

I'm using the constants on the `Response` class, but you can also use the status code numbers - e.g. `200` and `403`.

Running the updated tests

Running `vendor/bin/phpunit`, you should get two errors:

```
1) Drupal\Tests\atdc\Functional\ExampleTest::testFrontPage
Behat\Mink\Exception\ExpectationException: Current response status code is 200, but
403 expected.

2) Drupal\Tests\atdc\Functional\ExampleTest::testAdminPage
Behat\Mink\Exception\ExpectationException: Current response status code is 403, but
200 expected.

ERRORS!
Tests: 2, Assertions: 4, Errors: 2.
```

The responses are not returning the expected status codes, so the tests are failing.

If you're getting a 404 response on either test, check your `SIMPLETEST_BASE_URL` is correct and includes the port number. If it's incorrect, Drupal will send the requests to the wrong website.

Reviewing them, the front page should return a 200 response code (`HTTP_OK`) as it's accessible to all users, including anonymous users.

As you're logged out, the administration page should return a 403 (`HTTP_FORBIDDEN`).

Swapping the assertions should get the tests to pass.

Now, running `vendor/bin/phpunit` returns no errors or failures.

```
OK (2 tests, 4 assertions)
```

Congratulations!

Conclusion

In this chapter, you created a new Drupal 10 project, configured PHPUnit and created a custom module with your first passing browser tests.

From this, you can hopefully see that automated testing doesn't need to be difficult, and the configuration you've done here will work for the upcoming chapters, where you'll expand on what you've done and explore more that Drupal and PHPUnit have to offer.

Diving Deeper

At the end of the last chapter, we had a working test suite, with tests ensuring the correct response codes were returned from Drupal's front and administration pages.

So, how do we move on from here?

Testing as an authenticated user

In the current tests, we're testing the responses as an anonymous user.

So, how do we test as an authenticated user?

For example, how do we test the administration pages to see if they work for a user who's logged in?

Let's start with a new test:

```
public function testAdminPageLoggedIn(): void {
  $this->drupalGet('/admin');

  $assert = $this->assertSession();
  $assert->statusCodeEquals(Response::HTTP_OK);
}
```

This is the same test as before, only with a different expected status code. This time, we want a `200` status code, but as we're still anonymous, this test will fail.

Creating a user

Before we move on, an important thing to note is that these tests don't test against the data in your database.

A fresh installation from a `testing` profile is done for each test method, meaning each test is run against an empty Drupal site. This is why you didn't need to install Drupal in the first chapter and why it only needed to be running.

So, if you have existing users in your database, they won't be there to use.

This prevents contamination between tests, but you need to create the required data and environment for each test.

This is commonly known as the **Arrange** step of the test.

To create a user, use `$this->drupalCreateUser()` and `$this->drupalLogin()` to log in as that user.

```
$user = $this->drupalCreateUser();
```

```
$this->drupalLogin($user);
```

Now, we're no longer anonymous, though this test will still fail. Not all authenticated users can access the administration area.

To do this, a user needs two specific permissions - `access administration pages` and `administer site configuration`.

As we're testing against a temporary Drupal installation, we don't have access to any custom roles, so **we must add the permissions directly to the user instead of to a user role**.

To do this, when creating the user, include an array of permissions to add to it:

```
$user = $this->createUser(permissions: [
  'access administration pages',
  'administer site configuration',
]);
```

For readability, I like to add `permissions` as a named parameter, but it's optional.

With the permissions added, run `phpunit`, and the new test should pass.

What about content?

The same as users, we need to create any content we need in each test.

Let's create a page and test we can view it.

Firstly, let's ensure the page is not found:

```
public function testContent(): void {
  $this->drupalGet('/node/1');
  $this->assertSession()->statusCodeEquals(Response::HTTP_NOT_FOUND);
}
```

Similar to `$this->createUser()`, there are similar methods to create content types and nodes.

Again, as there are no existing content or content types, we need to create them and add the follow-up assertions:

```
public function testContent(): void {
  // ...

  $this->createContentType(['type' =>'page']);
  $this->createNode(['type' => 'page']);

  $this->drupalGet('/node/1');
```

```
  $this->assertSession()->statusCodeEquals(Response::HTTP_OK);
}
```

Enabling additional modules

You're probably expecting the test to pass now, but you'll likely get an error like this:

> Drupal\Core\Entity\Exception\NoCorrespondingEntityClassException: The Drupal\node\Entity\Node class does not correspond to an entity type.

To fix this, we need to tell Drupal to enable the `node` module within the test by adding this within the test class:

```
protected static $modules = ['node'];
```

If we need any additional modules, we can add those too.

Debugging

Here's a tip for today: if you're getting an unexpected status code or another error that you want to debug, you'll need to output the page content to see the error.

To do that, add this to your test, and it will output the page content:

```
var_dump($this->getSession()->getPage()->getContent());
```

Conclusion

In this chapter, you expanded your test suite to test as an authenticated user, and that content exists by creating a content type and node.

Next, we'll start building a new module from scratch.

Building a Blog

In the previous two chapters, you've created a Drupal project with PHPUnit and have a running test suite that uses users and content created as part of each test.

With what you've learned, let's create a simple Blog module with tests and test-driven development.

Creating the Blog page

First, let's create a page that will list the posts. This will be similar to our first tests for Drupal's front and admin pages.

Create a new `BlogPageTest` and have it extend `BrowserTestBase`.

Let's assert that a page should exist at `/blog` by returning a `200` status code, as this should be accessible by anonymous users.

```php
<?php

namespace Drupal\Tests\atdc\Functional;

use Drupal\Tests\BrowserTestBase;
use Symfony\Component\HttpFoundation\Response;

class BlogPageTest extends BrowserTestBase {

  protected $defaultTheme = 'stark';

  public function testBlogPage(): void {
    $this->drupalGet('/blog');

    $this->assertSession()->statusCodeEquals(Response::HTTP_OK);
  }

}
```

As you haven't created it, the status code should be a `404` - causing the test to fail.

> Tip: you can use `--filter testBlogPage` to run a single test or `--stop-on-failure` to stop running the tests as soon as an error occurs. These should shorten the time to run your tests, as you only run the tests you need.

Whilst you could create the page using the Views module, let's create a custom route.

Create an `atdc.routing.yml` file:

```yaml
# web/modules/custom/atdc/atdc.routing.yml
```

```yaml
atdc.blog:
  path: /blog
  defaults:
    _controller: Drupal\atdc\Controller\BlogPageController
    _title: Blog
  requirements:
    _permission: access content
```

With this added, the status code doesn't change and is a `404`.

Like in the previous chapter, you need to enable the `atdc` module by setting `$modules` in your test:

```php
protected static $modules = ['atdc'];
```

You'll also need to create an `atdc.info.yml` file so the module can be installed:

```yaml
# web/modules/custom/atdc/atdc.info.yml

name: ATDC
type: module
core_version_requirement: ^10
```

This should change the status code to a `403`, as you also need the `node` module for the `access content` permission:

```php
protected static $modules = ['node', 'atdc'];
```

This should cause the status code to change again - this time to a `500`.

This is progress.

The `atdc` module is being installed and enabled, and its routing file is being loaded. But, it references a Controller class that doesn't exist yet.

Let's do that next.

Creating a BlogPageController

Create the expected Controller class within a `src/Controller` directory:

```php
<?php

// web/modules/custom/atdc/src/Controller/BlogPageController.php

namespace Drupal\atdc\Controller;
```

```
class BlogPageController {

}
```

Note the namespace is different from the one within the test classes and we don't need to extend any other classes.

For this step, the simplest thing you can do to get a passing test is to return an empty render array.

As long as it's an array, even an empty one, the test should pass:

```
public function __invoke(): array {
  return [];
}
```

As a rule, you want the tests to pass as often and quickly as possible by doing the simplest thing to achieve it - even returning a hard-coded value or an empty array.

Now the test passes, you can add to it and drive out the next piece of functionality.

This is also a good time to do a `git commit`.

Asserting posts are visible

Again, let's start with a new test:

```
public function testPostsAreVisible(): void {
  // Arrange.
  $this->createNode(['type' => 'post', 'title' => 'First post']);
  $this->createNode(['type' => 'post', 'title' => 'Second post']);
  $this->createNode(['type' => 'post', 'title' => 'Third post']);

  // Act.
  $this->drupalGet('/blog');

  // Assert.
  $assert = $this->assertSession();
  $assert->pageTextContains('First post');
  $assert->pageTextContains('Second post');
  $assert->pageTextContains('Third post');
}
```

As we're returning an empty array within `BlogPageController`, the page will have no content and this test will fail with a message like:

> 1) Drupal\Tests\atdc\Functional\BlogPageTest::testPostsAreVisible

> Behat\Mink\Exception\ResponseTextException: The text "First post" was not found anywhere in the text of the current page.

Start by extending the `ControllerBase` base class within your Controller:

```
+ use Drupal\Core\Controller\ControllerBase;
+
- class BlogPageController {
+ class BlogPageController extends ControllerBase {
```

Now, within the `__invoke` method, add this to return a list of each node title:

```
public function __invoke(): array {
  $nodeStorage = $this->entityTypeManager()->getStorage('node');
  $nodes = $nodeStorage->loadMultiple();

  $build = [];

  $build['content']['#theme'] = 'item_list';

  foreach ($nodes as $node) {
    $build['content']['#items'][] = $node->label();
  }

  return $build;
}
```

As the node titles are within the page content, the test should pass.

To be confident, try returning an empty array again or removing the foreach loop, seeing the test fail, and reverting the change.

Confidence comes from tests that pass and fail when expected, so you're sure the correct behaviour is being tested, and the tests aren't passing accidentally.

You can add further tests, such as checking that only nodes of a specified node type are returned. Currently, all nodes would be listed, even if they aren't posts.

Asserting posts are in the correct order

We have a list of post titles on a page and a test to prove it, but what if we want to ensure the posts are shown in a specified order?

That's harder to do with a functional test, so in the next chapter, we'll refactor the code and look at Kernel tests.

Refactoring to a Repository

While the existing tests are passing, let's refactor the Controller and move the logic for loading posts into a new `PostNodeRepository` class.

Doing this will make the Controller simpler and cleaner and make it easier to test that posts are returned in the correct order.

Creating a PostNodeRepository

I like the Repository design pattern.

It's much better to have all logic to find and load nodes in one place instead of duplicating them across an application.

It also makes it easier to test.

To start, within your `atdc` module, create an `src/Repository` directory and a `PostNodeRepository.php` file inside it.

This will contain the `PostNodeRepository` class that will be responsible for loading the post nodes from the database instead of within `BlogPageController`.

Add this as the initial content:

```php
<?php

namespace Drupal\atdc\Repository;

use Drupal\node\NodeInterface;

final class PostNodeRepository {

  /**
   * @return array<int, NodeInterface>
   */
  public function findAll(): array {
    return [];
  }

}
```

I like to make my classes `final`, but this is optional and, by adding this docblock, we can specify the `findAll()` method should return an array of `NodeInterface` objects - making the code easier to read and providing better completion.

So far, you haven't changed `BlogPageController`, so the tests should still pass.

Next, let's move the logic for loading nodes into the Repository.

Moving the logic

Remove these lines from `BlogPageController`:

```
$nodeStorage = $this->entityTypeManager()->getStorage('node');
$nodes = $nodeStorage->loadMultiple();
```

Add them to the `findAll()` method, alter the first line that gets the `EntityTypeManager` (we'll refactor this later) and return the loaded nodes:

```
public function findAll(): array {
  $nodeStorage = \Drupal::entityTypeManager()->getStorage('node');
  $nodes = $nodeStorage->loadMultiple();

  return $nodes;
}
```

Within the `BlogPageController`, create a constructor method and inject the Repository using constructor property promotion:

```
public function __construct(
  private PostNodeRepository $postNodeRepository,
) {
}
```

Add `use Drupal\atdc\Repository\PostNodeRepository;` if needed, and use it to load the post nodes:

```
public function __invoke(): array {
  $nodes = $this->postNodeRepository->findAll();

  $build = [];

  $build['content']['#theme'] = 'item_list';
  foreach ($nodes as $node) {
    $build['content']['#items'][] = $node->label();
  }

  return $build;
}
```

We're almost back to a passing test, but there's more to do.

Getting back to green

Currently, the test is failing, as the response code is a `500` status because the `PostNodeRepository` isn't

being injected into the Controller.

It's expected within the constructor, but you must add a `create` method to inject it.

```
public static function create(ContainerInterface $container): self {
  return new self(
    $container->get(PostNodeRepository::class),
  );
}
```

You may also need to add `use Symfony\Component\DependencyInjection\ContainerInterface;` at the top of the file for the correct `ContainerInterface` to be used.

Creating a service

`$container→get()` uses a service's ID to retrieve it from the container, but `PostNodeRepository` isn't in the service container.

To do this, create an `atdc.services.yml` file within your module.

Add `PostNodeRepository` using the fully-qualified class name as the service name:

```
services:
  Drupal\atdc\Repository\PostNodeRepository:
    arguments: []
```

For now, include no arguments.

This should be enough to get the tests passing and back to green.

Injecting more dependencies

Before moving on, let's refactor the `PostNodeRepository` and inject the `EntityTypeManager` as a dependency.

The same as the `BlogPageController`, create a constructor method and inject the `EntityTypeManagerInterface`:

```
public function __construct(
  private EntityTypeManagerInterface $entityTypeManager,
) {
}
```

Add the `use Drupal\Core\Entity\EntityTypeManagerInterface;` if needed, and specify it as an argument so it's injected into the constructor:

```
services:
  Drupal\atdc\Repository\PostNodeRepository:
    arguments:
      - '@entity_type.manager'
```

Finally, update the code in the `findAll()` method:

```
- $nodeStorage = \Drupal::entityTypeManager()->getStorage('node');
+ $nodeStorage = $this->entityTypeManager->getStorage('node');
```

If this refactor is successful, the test will still pass.

Conclusion

Whilst we haven't added any new tests in this lesson, we've been able to use the existing tests to ensure that the functionality still works.

If you make a mistake, the tests will fail, and you can revert the changes and try again.

If they pass, the functionality still works as expected.

Now that the `PostNodeRepository` is in place, it'll be easier for us to test the order in which the posts are returned tomorrow.

Testing Post Ordering

Now we have the repository in place from the last lesson, let's move on to testing that the posts are returned in the correct order.

We want them to be returned based on their published date and not by their node ID or anything else.

To do this, we will use a different type of test - a Kernel test.

Introducing Kernel tests

So far, we've been using Functional (or Browser) tests to ensure the blog page exists and that the correct posts are displayed.

It's easy to assert the correct posts are shown on the page, but it's much harder to assert they're shown in the right order.

This is much easier to do with a Kernel test (aka. an integration test).

Instead of making HTTP requests and checking the responses, we can test the results from the repository and ensure it returns the results in the correct order.

Writing your first Kernel test

Let's create a new test that uses the `PostNodeRepository` to find the nodes and assert we get an expected number returned.

```php
<?php

namespace Drupal\Tests\atdc\Kernel;

use Drupal\atdc\Repository\PostNodeRepository;
use Drupal\KernelTests\Core\Entity\EntityKernelTestBase;

class PostNodeRepositoryTest extends EntityKernelTestBase {

  public function testPostsAreReturnedByCreatedDate(): void {
    // Arrange.

    // Act.
    $postRepository = $this->container->get(PostNodeRepository::class);
    assert($postRepository instanceof PostNodeRepository);
    $nodes = $postRepository->findAll();

    // Assert.
    self::assertCount(3, $nodes);
  }
```

```
}
```

As with the Functional test, the file and class name must have a `Test` suffix, and test methods should have a `test` prefix. As we're testing the `PostNodeRepository` class, the convention is to name the test `PostNodeRepositoryTest`.

As this is a Kernel test, it should be placed within the `tests/src/Kernel` directory and extend the `EntityKernelTestBase` class.

We could extend others, such as the regular `KernelTestBase`, but as we'll be working with nodes, `EntityKernelTestBase` is the better option.

Instead of making assertions based on the HTTP response, we're testing what's returned from the `findAll()` method.

Resolving setup test failures

Run the tests to see the first error:

```
Symfony\Component\DependencyInjection\Exception\ServiceNotFoundException: You have
requested a non-existent service "Drupal\atdc\Repository\PostNodeRepository".
```

Although you created the `PostNodeRepository` in the previous lesson and added it as a service, it's not found.

It's defined within `atdc.services.yml`, but we need to be explicit about which modules are enabled when running the test, and the `atdc` module isn't enabled.

Create a `$modules` array within the test class and add `atdc`:

```
protected static $modules = ['atdc'];
```

Run the tests again, and you should get a different error:

```
Drupal\Component\Plugin\Exception\PluginNotFoundException: The "node" entity type does
not exist.
```

As well as `atdc`, you must enable the `node` module. You can do so by adding it to the `$modules` array:

```
protected static $modules = ['node', 'atdc'];
```

Now, you should get a logic error instead of a setup error:

> Failed asserting that actual size 0 matches expected size 3.

Typically, Kernel tests have more setup steps, such as installing module configuration and creating specific database tables.

But, although they can be more complicated to set up, they're faster to run compared to Functional tests.

The type of test you pick will depend on what you're trying to test.

Creating posts

Currently, the 'Arrange' step within the test is empty, and whilst we're asserting there should be three posts returned, none are.

We need to create some posts within the test.

To access the `createNode()` method we've used in previous lessons within a Kernel test, you must use the `NodeCreationTrait`.

Add `use NodeCreationTrait` within the test class and `use Drupal\Tests\node\Traits\NodeCreationTrait;` as an import if needed.

Within the test, you can use `$this->createNode()` to create posts.

Create the three posts the test is expecting:

```
// Arrange.
$this->createNode(['type' => 'post']);
$this->createNode(['type' => 'post']);
$this->createNode(['type' => 'post']);
```

This should be enough for the test to pass.

Adding assertions for the order

Next, let's assert they're returned in a specific order.

Update the posts to have a specific title and created date so we can specify which order we expect them to be returned in and which titles they should have:

```
// Arrange.
$this->createNode([
  'created' => (new DrupalDateTime('-1 week'))->getTimestamp(),
  'title' => 'Post one',
  'type' => 'post',
]);

$this->createNode([
  'created' => (new DrupalDateTime('-8 days'))->getTimestamp(),
```

```
  'title' => 'Post two',
  'type' => 'post',
]);

$this->createNode([
  'created' => (new DrupalDateTime('yesterday'))->getTimestamp(),
  'title' => 'Post three',
  'type' => 'post',
]);
```

Note we're intentionally setting them to be in an incorrect order, to begin with, so the test doesn't pass accidentally. This way, we can see it fail and know the task is complete once it passes.

Next, assert that the titles are returned in the correct order.

```
self::assertSame(
  ['Post two', 'Post one', 'Post three'],
  array_map(
    fn (NodeInterface $node) => $node->label(),
    $nodes
  )
);
```

For each node in `$nodes`, get its label (title) and compare them with the titles in the order we want.

As expected, the test fails:

```
1) Drupal\Tests\atdc\Kernel\PostNodeRepositoryTest::testPostsAreReturnedByCreatedDate
Failed asserting that two arrays are identical.
--- Expected
+++ Actual
@@ @@
-     0 => 'Post two'
      1 => 'Post one'
-     2 => 'Post three'
+     2 => 'Post two'
+     3 => 'Post three'
 )
```

We want `Post two` to be returned first, followed by `Post one` and `Post three`.

Fixing the ordering

We need to update the code within `PostNodeRepository` to fix the ordering.

After loading the nodes, we need to sort them.

```php
public function findAll(): array {
  $nodeStorage = $this->entityTypeManager->getStorage('node');
  $nodes = $nodeStorage->loadMultiple();

  uasort($nodes, function (NodeInterface $a, NodeInterface $b): int {
    return $a->getCreatedTime() <=> $b->getCreatedTime();
  });

  return $nodes;
}
```

This sorts the nodes based on their created time in the desired order and returns them.

This gets us further, but the test is still failing.

Whilst the order is correct, the array keys don't match what we expect:

```
1) Drupal\Tests\atdc\Kernel\PostNodeRepositoryTest::testPostsAreReturnedByCreatedDate
Failed asserting that two arrays are identical.
--- Expected
+++ Actual
@@ @@
 Array &0 (
-    0 => 'Post two'
+    2 => 'Post two'
     1 => 'Post one'
-    2 => 'Post three'
+    3 => 'Post three'
 )
```

Finally, replace `return $nodes;` with `return array_values($nodes)` to reset the keys before returning them.

This should give us a passing test:

```
OK (1 test, 3 assertions)
```

And, because the correct titles are still being shown, our original Functional tests still pass, too:

```
OK (3 tests, 16 assertions)
```

Tip: to see the names of the tests in your output, add the `--testdox` flag to the `phpunit` command:

```
Blog Page (Drupal\Tests\atdc\Functional\BlogPage)
 ✔ Blog page
 ✔ Posts are visible
```

```
Post Node Repository (Drupal\Tests\atdc\Kernel\PostNodeRepository)
 ✔ Posts are returned by created date
```

Conclusion

In today's lesson, you learned about Kernel tests and wrote one to test the ordering of the posts returned from the `PostNodeRepository`.

Tomorrow, we'll refactor `PostNodeRepositoryTest` to use a `Builder` class and a custom assertion method.

Builders and custom assertions

In yesterday's lesson, you created your first Kernel test and used it to ensure the posts are returned from `PostNodeRepository` in the desired order.

This is how we're creating the posts currently:

```
$this->createNode([
  'created' => (new DrupalDateTime('-1 week'))->getTimestamp(),
  'title' => 'Post one',
  'type' => 'post',
]);

$this->createNode([
  'created' => (new DrupalDateTime('-8 days'))->getTimestamp(),
  'title' => 'Post two',
  'type' => 'post',
]);

$this->createNode([
  'created' => (new DrupalDateTime('yesterday'))->getTimestamp(),
  'title' => 'Post three',
  'type' => 'post',
]);
```

The Builder pattern is another design pattern I like, which makes it easier to build complex objects.

Let's create a Builder class to create the posts.

Creating a PostBuilder class

This is how I'd like to create a post using a `PostBuilder`:

```
PostBuilder::create()
  ->setCreatedDate('-1 week')
  ->setTitle('Post one')
  ->getPost();
```

This makes it easier to do by creating named methods for each value we want to set and not relying on array keys whilst also moving implementation details like using `DrupalDateTime` to set the `created` date.

To do this, create a new class at `src/Builder/PostBuilder.php`:

```
<?php

// web/modules/custom/atdc/src/Builder/PostBuilder.php
```

```php
namespace Drupal\atdc\Builder;

final class PostBuilder {

  public static function create(): self {
    return new self();
  }

}
```

It should be within the `Drupal\atdc\Builder` namespace and has a static `create` method that works as a named constructor and makes `PostBuilder::create()` work.

As it returns a new version of `self`, you can also chain methods onto it.

Add the additional methods and properties:

```php
private ?DrupalDateTime $created = NULL;

private string $title;

public function setCreatedDate(string $time = 'now'): self {
  $this->created = new DrupalDateTime($time);

  return $this;
}

public function setTitle(string $title): self {
  $this->title = $title;

  return $this;
}
```

Again, by returning `$this`, we can keep chaining methods.

Finally, create the `getPost()` method that creates the node based on the property values, saves it, and returns it.

```php
public function getPost(): NodeInterface {
  $post = Node::create([
    'created' => $this->created?->getTimestamp(),
    'title' => $this->title,
    'type' => 'post',
  ]);

  $post->save();

  return $post;
```

```
}
```

Now, refactor the test to use the `PostBuilder`:

```
PostBuilder::create()
  ->setCreatedDate('-1 week')
  ->setTitle('Post one')
  ->getPost();

PostBuilder::create()
  ->setCreatedDate('-8 days')
  ->setTitle('Post two')
  ->getPost();

PostBuilder::create()
  ->setCreatedDate('yesterday')
  ->setTitle('Post three')
  ->getPost();
```

Doing this simplifies the test and makes it easier to extend in the future by adding more methods to `PostBuilder`.

Creating a custom assertion

Finally, for today, let's refactor the assertion that verifies the titles are returned in the correct order.

This is the current assertion:

```
self::assertSame(
  ['Post two', 'Post one', 'Post three'],
  array_map(
    fn (NodeInterface $node) => $node->label(),
    $nodes
  )
);
```

We create an array of expected titles and compare that to an array created from `array_map`.

We can make this more reusable and readable by extracting this into a new custom assertion, which is just another static method.

Create a new static function at the bottom of the class with a name that describes what it's asserting:

```
/**
 * @param array<int, string> $expectedTitles
 * @param array<int, NodeInterface> $nodes
```

```
 */
private static function assertNodeTitlesAreSame(
  array $expectedTitles,
  array $nodes,
): void {
  self::assertSame(
    $expectedTitles,
    array_map(
      fn (NodeInterface $node) => $node->label(),
      $nodes
    )
  );
}
```

We can add arguments for the arrays of titles and nodes, and be explicit about what they contain by adding a docblock.

In this method, we can do the same logic and use `array_map` to create a list of node titles and compare them to the expected titles.

The benefits are that this now has a name that describes what we're asserting, and because it's a separate method, it can be reused in the same test or moved to a base class and used elsewhere.

Finally, refactor the test to use the new assertion:

```
self::assertNodeTitlesAreSame(
  ['Post two', 'Post one', 'Post three'],
  $nodes,
);
```

In my opinion, this is a lot better.

In tomorrow's lesson, let's add some more tests to the `PostNodeRepository` that we skipped in previous lessons.

Filling in old tests

In lesson 3, I mentioned that the current code has some gaps.

We checked the expected nodes were shown but not the opposite - the nodes we didn't expect to see weren't shown.

Let's fix that in this lesson.

Only returning published nodes

First, let's ensure that only published nodes are returned and displayed on the page.

We can do this easily with a functional test, so add a new test method to `BlogPostTest`:

```
public function testOnlyPublishedNodesAreShown(): void {
  PostBuilder::create()
    ->setTitle('Post one')
    ->isPublished()
    ->getPost();

  PostBuilder::create()
    ->setTitle('Post two')
    ->isNotPublished()
    ->getPost();

  PostBuilder::create()
    ->setTitle('Post three')
    ->isPublished()
    ->getPost();

  $this->drupalGet('/blog');

  $assert = $this->assertSession();
  $assert->pageTextContains('Post one');
  $assert->pageTextNotContains('Post two');
  $assert->pageTextContains('Post three');
}
```

Import the `PostBuilder` by adding `use Drupal\atdc\Builder\PostBuilder;` if needed, and run the test to see the first error:

```
Error: Call to undefined method Drupal\atdc\Builder\PostBuilder::isPublished()
```

In this test, we want to create some published and unpublished posts and assert only the published ones are shown, but we don't have this functionality on the `PostBuilder`.

To fix the error, add this function so it exists:

```
public function isPublished(): self {
  return $this;
}
```

We'll revisit this later once we have a failing test that requires further changes.

Running the tests again, you should get this unexpected error:

```
PDOException: SQLSTATE[23000]: Integrity constraint violation: 19 NOT NULL constraint
failed: node_field_data.created
```

When using `PostBuilder` in the previous lesson, we were always providing a created date, but, as we're not doing that in this test, the created date is `NULL`, causing this error.

Update the `getPost()` method to only set the created time if the `created` property has a value.

```
public function getPost(): NodeInterface {
  $post = Node::create([
    'title' => $this->title,
    'type' => 'post',
  ]);

  if ($this->created !== NULL) {
    $post->setCreatedTime($this->created->getTimestamp());
  }

  $post->save();

  return $post;
}
```

Now, we can see a similar error to the one before for `isNotPublished()`.

```
Error: Call to undefined method Drupal\atdc\Builder\PostBuilder::isNotPublished()
```

Again, create the simplest version of the method so the test can progress:

```
public function isNotPublished(): self {
  return $this;
}
```

Now, you should get the error you were likely expecting:

```
The text "Post two" appears in the text of this page, but it should not.
```

As we've set post two to be unpublished, we don't want it to be displayed.

However, we have no logic for that.

Updating PostBuilder

Within `PostBuilder`, we need to use the `isPublished` and `isNotPublished` methods to set the status of the node that's building built.

First, add an `isPublished` property to the class and set it to be `TRUE` by default:

```php
private bool $isPublished = TRUE;
```

Next, update the `isPublished()` and `isNotPublished()` methods to set the value appropriately:

```php
public function isNotPublished(): self {
  $this->isPublished = FALSE;

  return $this;
}

public function isPublished(): self {
  $this->isPublished = TRUE;

  return $this;
}
```

Even though `isPublished` is already true by default, doing this makes it explicit and makes what's being tested clearer.

Finally, within `getPost()`, update the code that creates the node to set the `status` property accordingly.

```php
$post = Node::create([
  'status' => $this->isPublished,
  'title' => $this->title,
  'type' => 'post',
]);
```

With these changes, the nodes have the correct status, but the test is still failing.

Updating PostNodeRepository

We also need to update the `PostNodeRepository` as that is responsible for loading and returning the relevant nodes from the database.

Currently, all we're doing is this:

```
$nodes = $nodeStorage->loadMultiple();
```

This will load all nodes, regardless of their type or status.

To fix this, change this to use `loadByProperties()` instead:

```
$nodes = $nodeStorage->loadByProperties();
```

`loadByProperties()` allows you to pass an array of properties and values to filter the results.

Note: you can also use `->getQuery()` if you prefer and write the query yourself.

For this case, let's add a property for `status` and its value to be `TRUE`:

```
$nodes = $nodeStorage->loadByProperties([
  'status' => TRUE,
]);
```

This ensures that only published nodes are returned, so the unpublished nodes are no longer shown, and the tests pass.

Only returning posts

The other issue is all published nodes are returned, even if they aren't posts.

Before adding this to `PostNodeRepository`, create a new failing test for it:

```
public function testOnlyPostNodesAreShown(): void {
  PostBuilder::create()->setTitle('Post one')->getPost();
  PostBuilder::create()->setTitle('Post two')->getPost();

  $this->createNode([
    'title' => 'This is not a post',
    'type' => 'page',
  ]);

  $this->drupalGet('/blog');

  $assert = $this->assertSession();
```

```
  $assert->pageTextContains('Post one');
  $assert->pageTextContains('Post two');
  $assert->pageTextNotContains('This is not a post');
}
```

Use `PostBuilder` to create two posts and `$this→createNode()` to create a post of a different type.

In this test, we want the two post titles to be shown but not the page's title.

If you run the test, it should fail as expected:

> The text "This is not a post" appears in the text of this page, but it should not.

Now we have a failing test, let's add the extra condition to `PostNodeRepository`:

```
$nodes = $nodeStorage->loadByProperties([
  'status' => TRUE,
  'type' => 'post',
]);
```

With both conditions, both tests should now pass, and you should only see published node articles on your blog page.

Conclusion

With these changes, the `PostNodeRepository` is more robust and fully featured.

While we could also write new Kernel tests for this functionality, it's already covered in the Functional tests. If you write accompanying Kernel tests, you wouldn't be able to make them fail without also making the Functional tests fail.

If you want to add them, you can.

It's up to you and your project team.

Tagging posts and test configuration

In this lesson, let's add tags to our posts using the `PostBuilder`.

As we're doing test-driven development, start by creating a new `PostBuilderTest`:

```
<?php

// web/modules/custom/atdc/tests/src/Kernel/Builder/PostBuilderTest.php

namespace Drupal\Tests\atdc\Kernel\Builder;

use Drupal\KernelTests\Core\Entity\EntityKernelTestBase;

final class PostBuilderTest extends EntityKernelTestBase {
}
```

As it's a Kernel test for a Builder class, place it within a `Kernel/Buider` directory and the equivalent namespace.

Testing the PostBuilder

Instead of writing test methods starting with `test`, you can also use an `@test` annotation or, in more recent versions of PHPUnit, a `#[Test]` attribute. This, with snake-case method names, is a popular approach as it can be easier to read.

Just be aware that if you write tests this way and don't add the annotation or attribute, the test won't be executed.

Let's start by testing the existing functionality within `PostBuilder` by verifying it returns published and unpublished posts.

Create these tests, which should pass by default as the code is already written:

```
/** @test */
public function it_returns_a_published_post(): void {
  $node = PostBuilder::create()
    ->setTitle('test')
    ->isPublished()
    ->getPost();

  self::assertInstanceOf(NodeInterface::class, $node);
  self::assertSame('post', $node->bundle());
  self::assertTrue($node->isPublished());
}

/** @test */
public function it_returns_an_unpublished_post(): void {
```

```php
  $node = PostBuilder::create()
    ->setTitle('test')
    ->isNotPublished()
    ->getPost();

  self::assertInstanceOf(NodeInterface::class, $node);
  self::assertSame('post', $node->bundle());
  self::assertFalse($node->isPublished());
}
```

In both tests, we create a new post node with `PostBuilder`, set a title and the appropriate status, get the post and assert it's in the correct state.

To verify the tests are working correctly, try changing some values in it and `PostBuilder` to see if it fails when expected.

Tagging posts

Next, create a test for adding tags to a post.

It should be mostly the same as the others, but instead of an assertion for the published status, try to use `var_dump()` to see the value of `field_tags`:

```php
/** @test */
public function it_returns_a_post_with_tags(): void {
  $node = PostBuilder::create()
    ->setTitle('test')
    ->getPost();

  self::assertInstanceOf(NodeInterface::class, $node);
  self::assertSame('post', $node->bundle());
  var_dump($node->get('field_tags'));
}
```

You should see an error message like this:

```
InvalidArgumentException: Field field_tags is unknown.
```

Why is this if `field_tags` is a default field that's created when Drupal is installed?

The same as users and content, the new Drupal instance is created for each test, which won't have your existing fields, so, in the test, you need to install the required configuration.

Creating a test module

To have the required configuration available, it can be added to the `atdc` module.

Any configuration files within a `config/install` directory can be installed, although if a site already has `field_tags` defined, we don't want to cause a conflict.

The convention is to create a test module that will only be required within the appropriate tests and to place the configuration there.

To do this, create a `web/modules/custom/atdc/modules/atdc_test` directory and an `atdc_test.info.yml` file with this content:

```
name: ATDC Test
type: module
core_version_requirement: ^10
hidden: true
```

Because of adding `hidden: true`, it won't appear in the modules list in Drupal's admin UI and avoid it being installed outside of the test environment.

Within the module, create a `config/install` directory, which is where the configuration files will be placed.

Creating configuration

But how do you know what to name the configuration files and what content to put in them?

Rather than trying to write them by hand, I create the configuration I need, such as fields, within a Drupal site and then export and edit the files I need.

To do that for this project, as I'm using the PHP built-in web server, I can use Drush to install Drupal using an SQLite database:

```
./vendor/bin/drush site:install --db-url sqlite://localhost/atdc.sqlite
```

Note: if you need Drush, run `composer require drush/drush` to install it via Composer.

If you have a database available, you can use that, too.

Once Drupal is installed and the configuration has been created, you can go to -/admin/config/development/configuration/single/export and select the configuration type and name.

The filename is shown at the bottom of the page, and you can copy the content into files within your module.

The `uuid` and `_core` values are site-specific, so they can be removed.

To add `field_tags`, you'll need both the field and field storage configuration.

These are the files that I created in my module based on the field I created.

`field.field.node.post.field_tags.yml`:

```
langcode: en
status: true
dependencies:
  config:
    - field.storage.node.field_tags
    - node.type.post
    - taxonomy.vocabulary.tags
id: node.post.field_tags
field_name: field_tags
entity_type: node
bundle: post
label: Tags
description: 'Enter a comma-separated list. For example: Amsterdam, Mexico City,
"Cleveland, Ohio"'
required: false
translatable: true
default_value: {  }
default_value_callback: ''
settings:
  handler: 'default:taxonomy_term'
  handler_settings:
    target_bundles:
      tags: tags
    sort:
      field: _none
    auto_create: true
field_type: entity_reference
```

`field.storage.node.field_tags.yml`:

```
langcode: en
status: true
dependencies:
  module:
    - node
    - taxonomy
id: node.field_tags
field_name: field_tags
entity_type: node
type: entity_reference
settings:
  target_type: taxonomy_term
module: core
locked: false
cardinality: -1
translatable: true
indexes: {  }
```

```
persist_with_no_fields: false
custom_storage: false
```

Then, enable the module within `PostBuilderTest`:

```
protected static $modules = [
  // Core.
  'node',

  // Custom.
  'atdc_test',
];
```

Finally, install the configuration to create the field. Add this within the test:

```
$this->installConfig(modules: [
  'atdc_test',
]);
```

After adding this and attempting to install the configuration to add the field, you'll get an error:

```
Exception when installing config for module atdc_test, the message was: Field
'field_tags' on entity type 'node' references a target entity type 'taxonomy_term',
which does not exist.
```

Fixing setup issues

The test is trying to install the `field_tags`, but it's missing a dependency. Tags reference a taxonomy term, but we haven't enabled the Taxonomy module within the test.

Enable the `taxonomy` module by adding it to the `$modules` array, and the error should change:

```
Exception when installing config for module atdc_test, message was: Missing bundle
entity, entity type node_type, entity id post.
```

As well as the field configuration, we also need to create the Post content type.

This can be done by creating a `node.type.post.yml` file:

```
langcode: en
status: true
dependencies: {  }
name: Post
type: post
```

```
description: ''
help: ''
new_revision: true
preview_mode: 1
display_submitted: true
```

With this configuration, `field_tags` should be created on the Post content type, which is enough for the current test to pass.

Setting tags

Let's update the test and add assertions about the tags being saved and returned.

Get the tags from the post and assert that three tags are returned:

```
$tags = $node->get('field_tags')->referencedEntities();
self::assertCount(3, $tags);
```

As none have been added, this would fail the test.

Update the test to use a `setTags()` method that you haven't created yet:

```
$node = PostBuilder::create()
  ->setTitle('test')
  ->setTags(['Drupal', 'PHP', 'Testing'])
  ->getPost();
```

You should get an error confirming the method is undefined:

```
Error: Call to undefined method Drupal\atdc\Builder\PostBuilder::setTags()
```

To fix this, add the `tags` property and `setTags()` method to `PostBuilder`:

```
/**
 * @var string[]
 */
private array $tags = [];

/**
 * @param string[] $tags
 */
public function setTags(array $tags): self {
  $this->tags = $tags;

  return $this;
```

```
}
```

Tags will be an array of strings, and `setTags()` should set the tags to the `tags` property.

Next, add the logic to `getPost()` to create a taxonomy term for each tag name.

```
$tagTerms = [];

if ($this->tags !== []) {
  foreach ($this->tags as $tag) {
    $term = Term::create([
      'name' => $tag,
      'vid' => 'tags',
    ]);

    $term->save();

    $tagTerms[] = $term;
  }

  $post->set('field_tags', $tagTerms);
}
```

If `$this->tags` is not empty, create a new taxonomy term for each one and save it to the post.

Base table or view not found

When running the test, you may see an error similar to this:

```
Base table or view not found: 1146 Table 'app.test95351694taxonomy_term_data' doesn't
exist
```

Or:

```
General error: 1 no such table: app.taxonomy_term_data
```

The error may depend on the Drupal version or database type used, but is because the table doesn't exist within the test database.

Because this is a Kernel test, we need to be explicit about which tables are installed.

To install the table add this line to the top of the test method:

```
$this->installEntitySchema('taxonomy_term');
```

This will install the table and ensure it's present to save data to.

Adding tag assertions

As well as asserting we have the correct number of tags, let's also assert that the correct tag names are returned and that they are the correct type of term.

```
self::assertContainsOnlyInstancesOf(TermInterface::class, $tags);

foreach ($tags as $tag) {
  self::assertSame('tags', $tag->bundle());
}
```

To assert the tags array only includes taxonomy terms, use `self::assertContainsOnlyInstancesOf()`, and to check each term has the correct term type, loop over each term and use `self::assertSame()`.

Next, add some new assertions to the test to check the tag names match the specified tags.

```
self::assertSame('Drupal', $tags[0]->label());
self::assertSame('PHP', $tags[1]->label());
self::assertSame('Testing', $tags[2]->label());
```

These should pass as we already have code for them, but to see them fail, try changing a term type or tag name in the assertion or when creating the post to ensure the test works as expected.

Conclusion

In this lesson, you learned how to add the configuration required for tests by creating a custom test module with the required configuration and how to install it within the test so configuration, such as fields, are available.

You created `PostBuilderTest` - a Kernel test for testing `PostBuilder`.

In the next lesson, we'll look at unit testing and start to wrap up this course.

Introducing Unit Tests

Unit tests are the last type of test we'll cover in this course.

Similar to Kernel tests, in a Unit test, there is no browser to make HTTP requests with, but also no database or service container, so everything needs to be created from scratch.

I do outside-in testing and start with Functional and Kernel tests, so I don't tend to write many Unit tests.

I prefer to use real objects as opposed to mocks and have seen tests that create mocks and only test the mock and not the rest of the code.

I've also seen Unit tests that are very tightly coupled to the implementation, such as asserting a method is only called a certain number of times. This makes the code harder to refactor and could result in a test failing when its functionality is working.

Your first Unit test

Based on what you've learned so far, let's write a Unit test that we'd expect to pass:

```php
<?php

namespace Drupal\Tests\atdc\Unit;

use Drupal\node\Entity\Node;
use Drupal\node\NodeInterface;
use Drupal\Tests\UnitTestCase;

final class PostWrapperTest extends UnitTestCase {

  /** @test */
  public function it_wraps_a_post(): void {
    $node = Node::create(
      entity_type: 'post',
      values: [],
    );

    self::assertInstanceOf(NodeInterface::class, $node);
  }

}
```

This test is within the `tests/src/Unit` directory and the equivalent namespace and extends the `UnitTestCase` class.

However, when you run the test, you'll get an error:

```
Drupal\Core\DependencyInjection\ContainerNotInitializedException: \Drupal::$container
is not initialized yet. \Drupal::setContainer() must be called with a real container.
```

In a Unit test, there is no database or service container, so you need to use mocks instead.

Update the test to create a mock version of `NodeInterface` instead.

As the mock an instance of `NodeInterface`, it satisfies the assertion and the test passes.

```
/** @test */
public function it_wraps_a_post(): void {
  $node = $this->createMock(NodeInterface::class);

  self::assertInstanceOf(NodeInterface::class, $node);
}
```

Next, add an assertion to ensure the bundle is correct:

```
self::assertSame('post', $node->bundle());
```

This will fail with this error:

```
Failed asserting that null is identical to 'post'.
```

Because you're using a mock, all methods will return `NULL`.

To get this to pass, you need to define what `$this→bundle()` will return:

```
$node->method('bundle')->willReturn('post');
```

However, this leads us to the situation I described, where you're only testing what's defined in the mock and not any valuable logic.

Let's improve this by introducing a `PostWrapper`.

Wrapping posts

Let's create a `PostWrapper` class that wraps a post node and has some methods that return specific values from it.

Within the test, instantiate a new `Postwrapper` class that takes the node as an argument.

Then, add an assertion that a `getType()` method should return `post`.

```
$wrapper = new PostWrapper($node);

self::assertSame('post', $wrapper->getType());
```

Next, create a `PostWrapper` class with the `getType()` method:

```
<?php

namespace Drupal\atdc;

use Drupal\node\NodeInterface;

final class PostWrapper {

  public function __construct(private NodeInterface $post) {
  }

  public function getType(): string {
    return $this->post->bundle();
  }

}
```

Now the test isn't testing the mock data directly, but the mock data is used within the `PostWrapper` to assert it is returning the expected value.

Not wrapping a page

We've tested that the `PostWrapper` works with post nodes, but let's also ensure it won't work with other node types.

Create a new test that creates a mock node and returns `page` as the bundle:

```
/**
 * @test
 * @testdox It can't wrap a page
 */
public function it_cant_wrap_a_page(): void {
  self::expectException(\InvalidArgumentException::class);

  $node = $this->createMock(NodeInterface::class);
  $node->method('bundle')->willReturn('page');

  new PostWrapper($node);
}
```

Before creating a new `PostWrapper`, assert that an `InvalidArgumentException` should be thrown. As no

assertion is thrown, this test should fail:

```
Failed asserting that exception of type "InvalidArgumentException" is thrown.
```

To fix it, within the constructor for `PostWrapper`, check the bundle and throw the expected Exception if the bundle is not `post`:

```
/**
 * @throws \InvalidArgumentException
 */
public function __construct(private NodeInterface $post) {
  if ($post->bundle() !== 'post') {
    throw new \InvalidArgumentException();
  }
}
```

Again, instead of making assertions against the mock data directly, it's used to provide known data to the classes that need it.

Conclusion

In this lesson, I introduced unit testing and mocking.

In tomorrow's lesson, the final one in this course, I'll show you an example of how to use mocks with Service classes.

Mocking services

In this final lesson, let's continue looking at unit testing, mocking and how we can mock Drupal's services that are dependencies for our classes.

In lesson 5, you used Kernel tests to ensure the correct posts were returned from `PostNodeRepository` and in the correct order.

Let's see how that would look as a unit test.

Creating the test

Create a new test, `PostNodeRepositoryUnitTest` and, for now, just create a new `PostNodeRepository`:

```
<?php

// web/modules/custom/atdc/tests/src/Unit/PostNodeRepositoryUnitTest.php

final class PostNodeRepositoryUnitTest extends UnitTestCase {

  /** @test */
  public function it_returns_posts(): void {
    $repository = new PostNodeRepository();
  }

}
```

Running the test will give this error:

```
ArgumentCountError: Too few arguments to function
Drupal\atdc\Repository\PostNodeRepository::__construct(), 0 passed
```

This is expected as `PostNodeRepository` has a dependency - the `EntityTypeManager`.

But, as this is a unit test, you can't get the Repository from the service container, and you need to instantiate it as well as any dependencies.

Try to fix this by creating a new `EntityTypeManager` and injecting it into the constructor:

```
$repository = new PostNodeRepository(
  new EntityTypeManager(),
);
```

Running the tests again will give you a similar error:

```
ArgumentCountError: Too few arguments to function
```

```
Drupal\Core\Entity\EntityTypeManager::__construct(), 0 passed
```

`EntityTypeManager` also has dependencies that need to be injected, and they may have dependencies.

Instead of doing this manually, let's start using mocks.

Adding the first mock

Add `use Drupal\Core\Entity\EntityTypeManagerInterface;` and create a mock to use instead of the manually created version.

```
$repository = new PostNodeRepository(
  $this->createMock(EntityTypeManagerInterface::class),
);
```

As the mock implements `EntityTypeManagerInterface`, this will fix the failure, and the test will continue.

Getting the posts

Next, try to get the posts from the Repository:

```
$repository->findAll();
```

Instead of returning a result, this also results in an error:

```
Error: Call to a member function loadByProperties() on null
```

Within `PostNodeRepository`, we use the `getStorage()` on `EntityTypeManager` to get the node storage, which is an instance of `EntityStorageInterface`.

For the test to work, this needs to be mocked too and returned from the `getStorage()` method.

Create a mock of `EntityStorageInterface`, which will be used as the node storage:

```
$nodeStorage = $this->createMock(EntityStorageInterface::class);
```

Next, this needs to be returns from the mock `EntityTypeManager`.

To do this, specify that the `getStorage()` method when called with the value `node`, will return the mocked node storage:

```
$entityTypeManager = $this->createMock(EntityTypeManagerInterface::class);
$entityTypeManager->method('getStorage')->with('node')->willReturn($nodeStorage);
```

```
$repository = new PostNodeRepository($entityTypeManager);
```

This will then be returned instead of `NULL` and fix the error.

Creating nodes and adding assertions

Next, let's create and return the nodes we need and add the assertions.

You'll need to use a mock for each node and set what each method needs to return.

The same as the Kernel test, set a title for each post with different created times.

```
$node1 = $this->createMock(NodeInterface::class);
$node1->method('bundle')->willReturn('post');
$node1->method('getCreatedTime')->willReturn(strtotime('-1 week'));
$node1->method('label')->willReturn('Post one');

$node2 = $this->createMock(NodeInterface::class);
$node2->method('bundle')->willReturn('post');
$node2->method('getCreatedTime')->willReturn(strtotime('-8 days'));
$node2->method('label')->willReturn('Post two');

$node3 = $this->createMock(NodeInterface::class);
$node3->method('bundle')->willReturn('post');
$node3->method('getCreatedTime')->willReturn(strtotime('yesterday'));
$node3->method('label')->willReturn('Post three');
```

Then, specify the `loadByProperties` method should return the posts.

```
$nodeStorage->method('loadByProperties')->willReturn([
  $node1,
  $node2,
  $node3,
]);
```

Finally, add some assertions that the nodes returned are the correct ones and in the correct order:

```
$posts = $repository->findAll();

self::assertContainsOnlyInstancesOf(NodeInterface::class, $posts);

$titles = array_map(
  fn (NodeInterface $node) => $node->label(),
  $posts,
);
```

```
self::assertCount(3, $titles);
self::assertSame(
  ['Post two', 'Post one', 'Post three'],
  $titles,
);
```

As the assertions should match the returned values, this test should now pass.

This is testing the same thing as the kernel test, but it's your preference which way you prefer.

Conclusion

Hopefully, if you run your whole testsuite, you should see output like this:

```
PHPUnit 9.6.15 by Sebastian Bergmann and contributors.

.........                                                       9 / 9 (100%)

Time: 00:07.676, Memory: 10.00 MB
```

Or, if you use `--testdox`, output like this:

```
PHPUnit 9.6.15 by Sebastian Bergmann and contributors.

Blog Page (Drupal\Tests\atdc\Functional\BlogPage)
 ✔ Blog page
 ✔ Posts are visible
 ✔ Only published nodes are shown
 ✔ Only post nodes are shown

Post Builder (Drupal\Tests\atdc\Kernel\Builder\PostBuilder)
 ✔ It returns a published post
 ✔ It returns an unpublished post
 ✔ It returns a post with tags

Post Node Repository (Drupal\Tests\atdc\Kernel\PostNodeRepository)
 ✔ Posts are returned by created date

Post Node Repository Unit (Drupal\Tests\atdc\Unit\PostNodeRepositoryUnit)
 ✔ It returns posts

Time: 00:07.097, Memory: 10.00 MB

OK (9 tests, 71 assertions)
```

Everything should be passing, and your testsuite should have a combination of different types of tests.

In this course, you've learned:

- How to configure Drupal and PHPUnit to run automated tests.
- How to write functional, kernel and unit tests.
- How to create data, such as node types, content and users within tests.
- How to manage configuration using test-specific modules.
- How to write unit tests and use mocks.
- Some small PHP tips and tricks, such as promoted constructor properties and the `@test` and `@testdox` parameters in PHPUnit.

I couldn't cover everything in a short email course, but I hope it was useful.

Questions and feedback

Thank you for taking my Introduction to Automated Testing in Drupal email course.

I'd appreciate any feedback, so if you wouldn't mind, press reply and let me know what you thought of the course.

Also, I'd love to know your next steps are and what I can do to help.

You can register for my [Daily Email list][daily] to get daily software development emails and updates about future products and courses or see when the next date is for my [online Drupal testing workshop][dto].

I also offer private workshops and talks for development teams, [1-on-1 consulting calls][call] and [pair programming sessions][pair], [development team coaching][team] and [Drupal development subscriptions][subscription].

Happy testing!

Oliver