CSY2028 Web Programming Topic 16

Tom Butler
thomas.butler@northampton.ac.uk

Topic 16

- Quick recap of last week
- Automated testing continued

Form Validation

- The follow code could be run when a form is submitted:

```
if (isset($_POST['submit'])) {

    $valid = false;

    if ($_POST['user']['firstname'] == '') {
        $valid = false;
    }

    if ($_POST['user']['surname'] == '') {
        $valid = false;
    }

    if ($_POST['user']['email'] == '') {
        $valid = false;
    }

    if ($valid) {
        $users->save($_POST['user']);
    }

    $user = $_POST['user'];
}
```

Testing

- To test this code you must open up the page, enter information into the form and then press submit
- You can then check the response is the one you expected (E.g. the correct error messages showing)

Testing

- To fully test this code you need to test:
  - Leaving every field blank
    - Checking the error is shown
  - Leaving firstname blank
    - Checking the error is shown
  - Leaving surname blank
    - Checking the error is shown
  - Leaving email blank
    - Checking the error is shown
  - Leaving two of the three fields blank
    - Checking the error is shown
  - Filling in every field
    - Checking the record has been inserted into the databases

Testing

- To fully test the code requires a significant amount of tests
- If a problem is discovered in one of the tests a fix can be applied
- However, how do you know that the fix hasn't broken one of the other tests?
- The only way to do this is to run all the tests again!

Testing

- Each test requires:
  - Opening the page in the browser
  - Filling in the form fields
  - Pressing submit
  - Checking the result
- This must be done for each of the tests above

Automated Testing

- One way to overcome this is *automated testing*
- An automated test is a test that is written in *code* to perform a specific task
- The test code performs the task and checks the result

Automated testing

- However, to write automated tests you need to set up your code so that it can be tested
- If the code is linked to $_POST, variables can only come from $_POST. This means the test requires a form submission to work:

```php
if (isset($_POST['submit'])) {

    $valid = false;

    if ($_POST['user']['firstname'] == '') {
        $valid = false;
    }

    if ($_POST['user']['surname'] == '') {
        $valid = false;
    }

    if ($_POST['user']['email'] == '') {
        $valid = false;
    }

    if ($valid) {
        $users->save($_POST['user']);
    }

    $user = $_POST['user'];
}
```

Automated testing

- The code you want to test also needs to be isolated from the rest of the code
  - You only want to test that the form submission works as expected
  - Note that the template is displayed as expected

Automated testing

- To fix both the problems:
  - Isolating the code you want to test
  - Removing the reliance on a form submission (the $_POST variable)
- The code you want to test can be moved to a function
- All variables from $_POST can be passed as arguments instead of reading from $_POST directly

```php
<?php
//Include the file that contains the loadTemplate function
require 'loadTemplate.php';

require 'database.php';

$users = new DatabaseTable($pdo, 'users');

if (isset($_POST['submit'])) {
        $saved = saveUser($users, $_POST['user']);

        $user = $_POST['user'];
}
else {
        if (isset($_GET['id'])) {
                $user = $users->find('id', $_GET['id']);
        }
        else $user = false;
}

$content = loadTemplate('account.php', [
        'user '=> $user
]);

$templateVars = [
  'title' => 'About our company',
  'content' => $content
];

echo loadTemplate('layout.php', $templateVars);
```

```php
function saveUser($users, $user) {
    $valid = false;

    if ($user['firstname'] == '') {
        $valid = false;
    }

    if ($user['surname'] == '') {
        $valid = false;
    }

    if ($user['email'] == '') {
        $valid = false;
    }

    if ($valid) {
        $users->save($user);
    }

    return $valid;
```

Testing

- By separating out the code you want to test the code can be run without needing the form to be filled in and submitted

Automated Testing

- It's now possible to test with some dummy data that checks the function returns the expected result

```
$invalidEmail = [
    'firstname' => 'John',
    'surname' => 'Smith',
    'email' => ''
];


$valid = saveUser($users, $invalidEmail);

//This is supposed to be invalid
if ($valid == false) {
    echo 'Test Passed';
}
else {
    echo 'Test failed';
}
```

Automated Testing

- Tests can then be added for all possible outcomes:

```php
$invalidFirstname = [
    'firstname' => '',
    'surname' => 'Smith',
    'email' => 'john@example.org'
];


$valid = saveUser($users, $invalidFirstname);

//This is supposed to be invalid
if ($valid == false) {
    echo 'Test Passed';
}
else {
    echo 'Test failed';
}
```

```php
$invalidSurname = [
    'firstname' => 'John',
    'surname' => '',
    'email' => 'john@example.org'
];


$valid = saveUser($users, $invalidSurname);

//This is supposed to be invalid
if ($valid == false) {
    echo 'Test Passed';
}
else {
    echo 'Test failed';
}
```

PHPUnit

- Automated Testing is a very common problem that has to be solved
- As with most common problems there are existing solutions
- For PHP there is a library called PHPUnit which allows you to quickly write automated tests and includes tools to give you better information about why a test failed

A Basic test

- Once you have created a test file, e.g. SaveUserTest.php you can write a test
- The structure of a test file looks like this:
- The name of the class should match the name of the file (without the .php extension)

```php
<?php

class SaveUserTest extends \PHPUnit\Framework\TestCase {

}
```

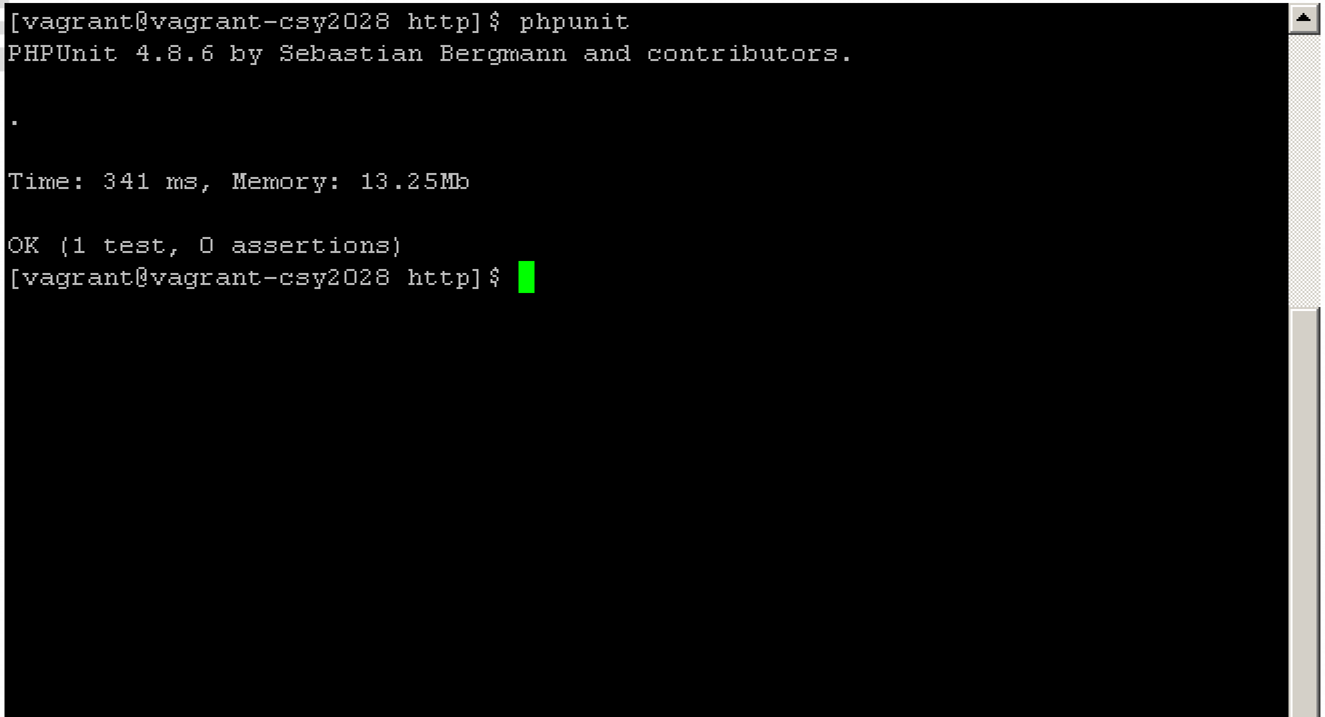
A Basic test

- Each test is a function inside the class
- To create a test add a public function with a name starting `test` e.g.

```
<?php

class SaveUserTest extends \PHPUnit\Framework\TestCase {
  public function testInvalidEmail() {

  }
}
```

A Basic test

- Running phpunit again will now produce the output:



```
[vagrant@vagrant-csy2028 http]$ phpunit
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

.

Time: 341 ms, Memory: 13.25Mb

OK (1 test, 0 assertions)
[vagrant@vagrant-csy2028 http]$
```

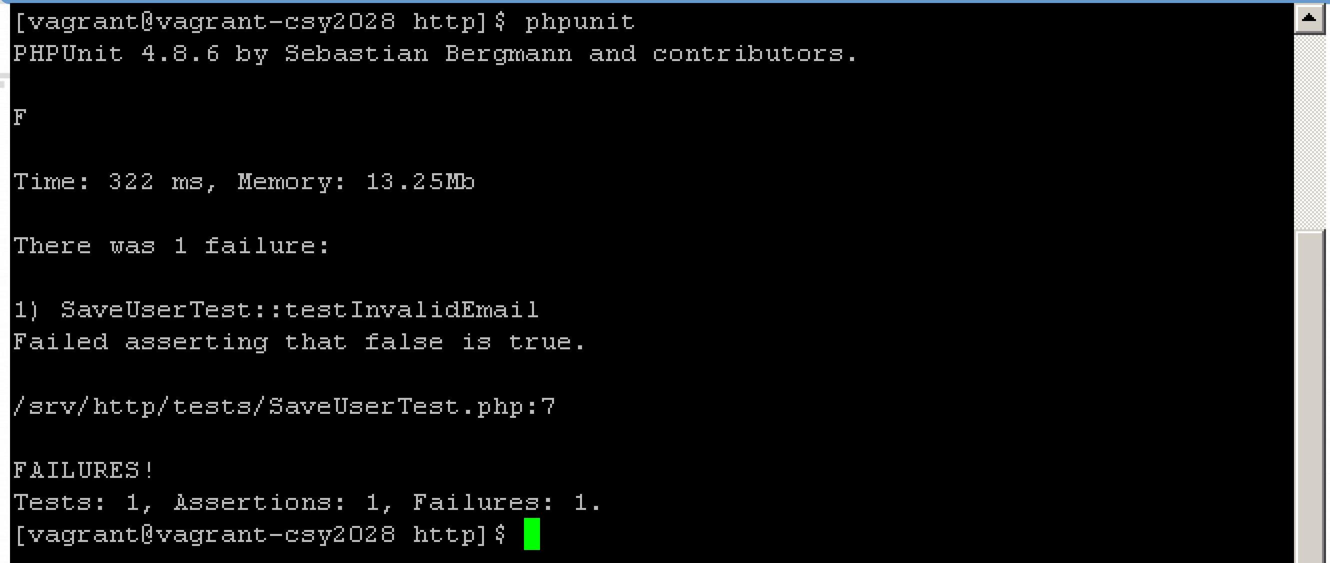
A Basic test

- This now shows that 1 test has been run
- As there is no code nothing is actually being tested
- Each test should feature one of more *assertions*
- An assertion is a special type of check
- E.g. checking a value is true or false

A Basic test

- To test a value is true you can use this:

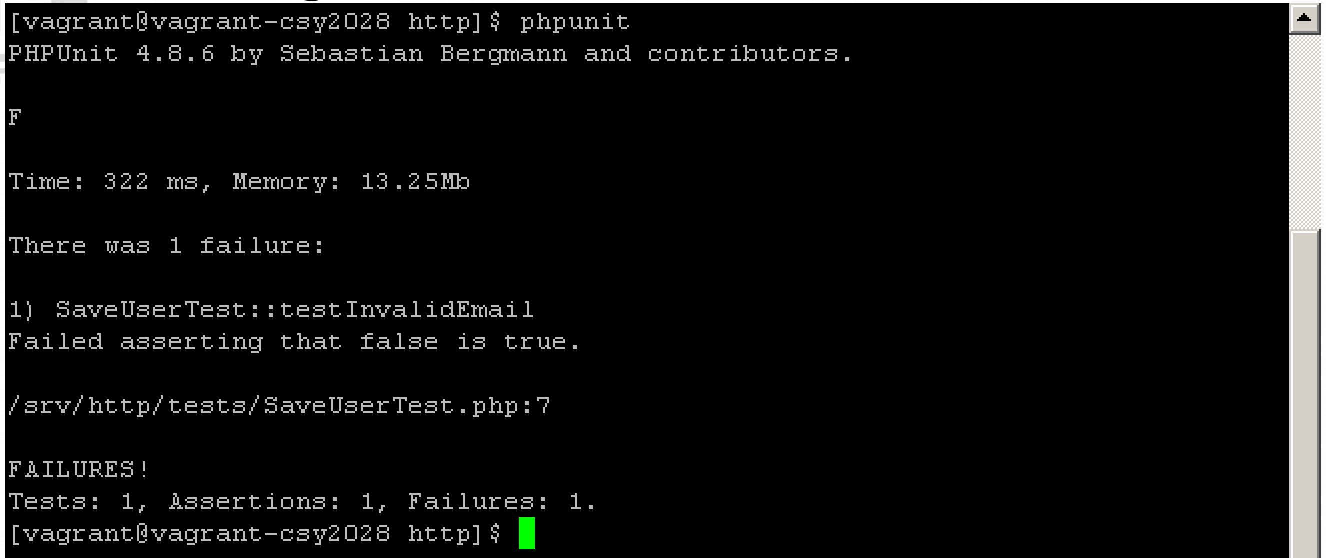
```php
<?php

class SaveUserTest extends \PHPUnit\Framework\TestCase {
  public function testInvalidEmail() {
    $variable = false;

    $this->assertTrue($variable);
  }
}
```



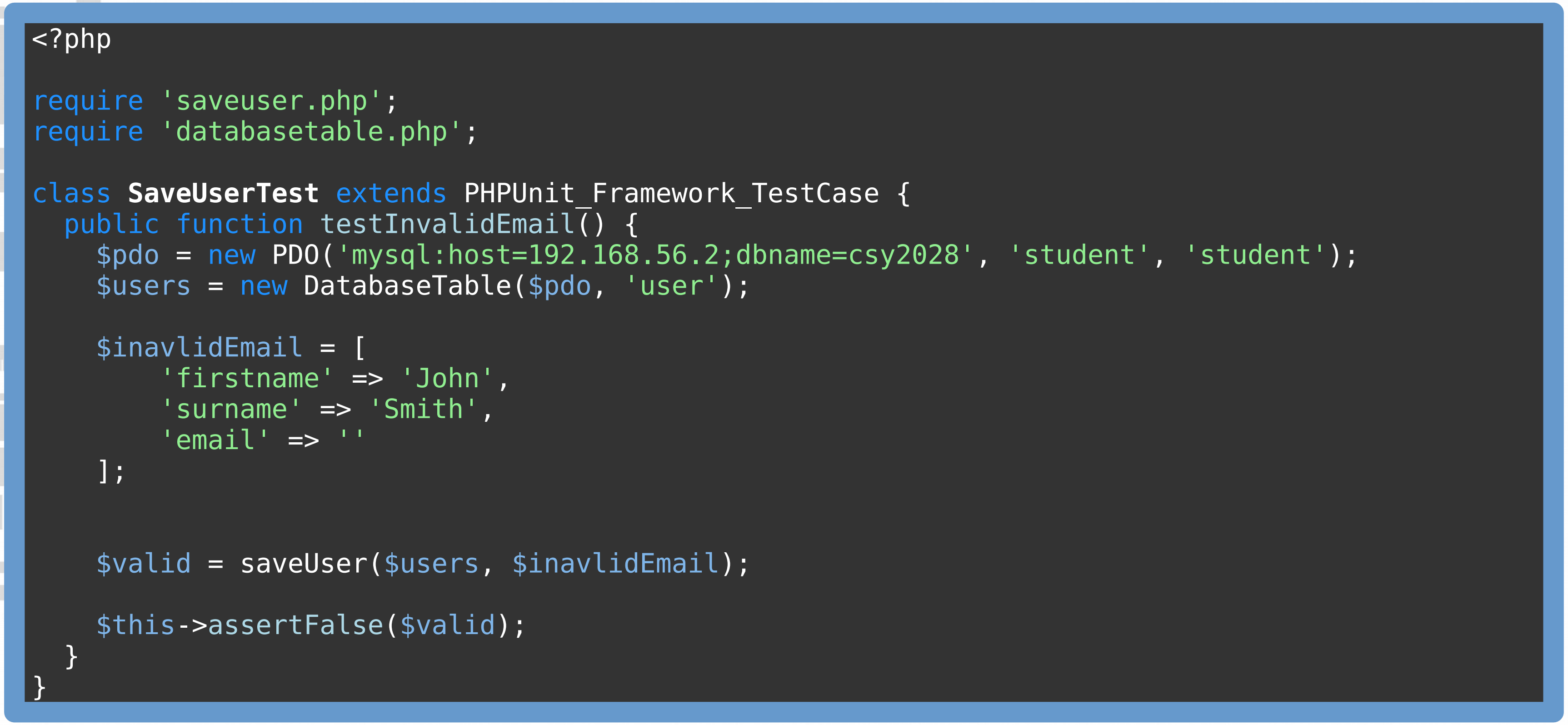
```
[vagrant@vagrant-csy2028 http]$ phpunit
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

F

Time: 322 ms, Memory: 13.25Mb

There was 1 failure:

1) SaveUserTest::testInvalidEmail
Failed asserting that false is true.

/srv/http/tests/SaveUserTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
[vagrant@vagrant-csy2028 http]$
```

PHPUnit

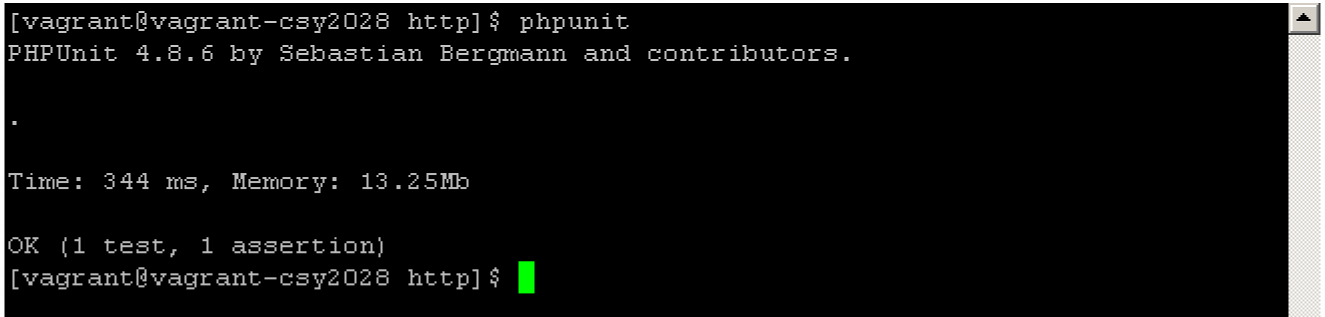
- When you make an assertion PHPunit will check to see whether the variable is true.
- If it's not it displays an error saying which test failed and why it failed (e.g. false should be true)



```
[vagrant@vagrant-csy2028 http]$ phpunit
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

F

Time: 322 ms, Memory: 13.25Mb

There was 1 failure:

1) SaveUserTest::testInvalidEmail
Failed asserting that false is true.

/srv/http/tests/SaveUserTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
[vagrant@vagrant-csy2028 http]$
```



```
<?php

require 'saveuser.php';
require 'databasetable.php';

class SaveUserTest extends PHPUnit_Framework_TestCase {
  public function testInvalidEmail() {
    $pdo = new PDO('mysql:host=192.168.56.2;dbname=csy2028', 'student', 'student');
    $users = new DatabaseTable($pdo, 'user');

    $inavlidEmail = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => ''
    ];


    $valid = saveUser($users, $inavlidEmail);

    $this->assertFalse($valid);
  }
}
```



```
[vagrant@vagrant-csy2028 http]$ phpunit
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

.

Time: 344 ms, Memory: 13.25Mb

OK (1 test, 1 assertion)
[vagrant@vagrant-csy2028 http]$
```

Extra tests

- You can add as many tests to the class as you like by adding more functions
- For example, testing that the function accepts a missing email address:

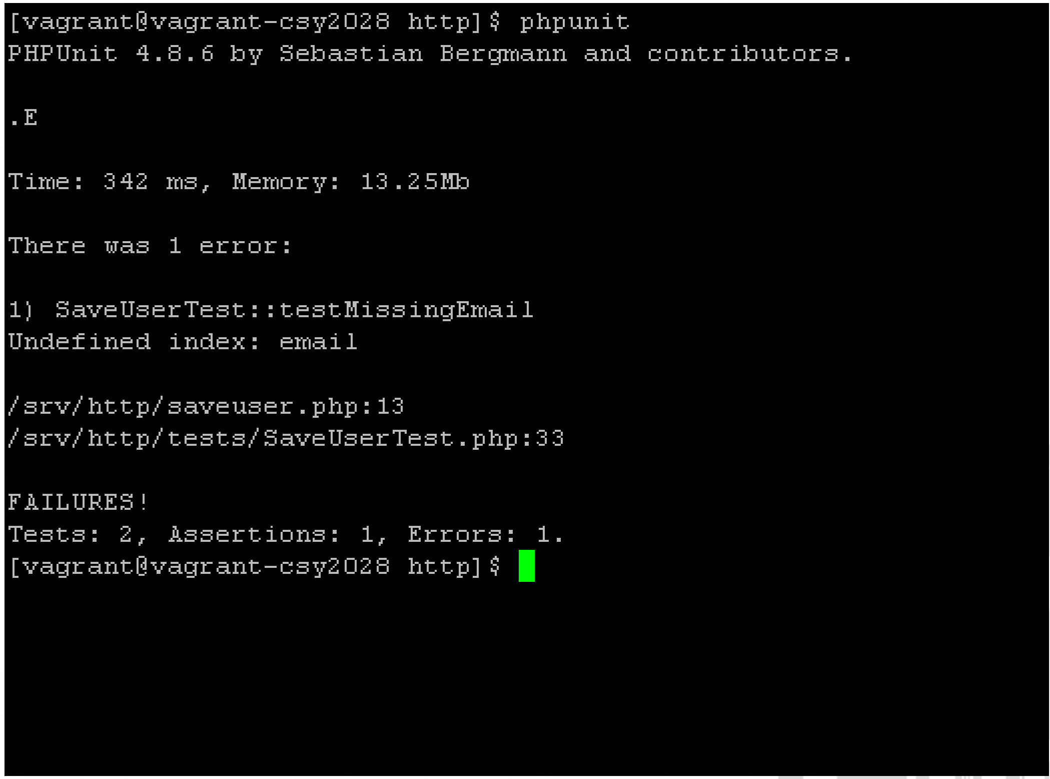
PHPUnit setUp

- PHPUnit will automatically look for a function named setUp() and call it before running each test
- This can be used to remove duplication
- Like any class, you can create a class variable that is available in every function

```php
<?php
require 'saveuser.php';
require 'databasetable.php';

class SaveUserTest extends PHPUnit_Framework_TestCase {
  private $users;

  public function setUp() {
    $pdo = new PDO('mysql:host=192.168.56.2;dbname=csy2028', 'student', 'student');
      $this->users = new DatabaseTable($pdo, 'user');
  }

  public function testInvalidEmail() {
    $invalidEmail = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => ''
    ];


    $valid = saveUser($this->users, $invalidEmail);

    $this->assertFalse($valid);
  }

  public function testMissingEmail() {
    $inavlidEmail = [
        'firstname' => 'John',
        'surname' => 'Smith',
    ];

    $valid = saveUser($this->users, $inavlidEmail);

    $this->assertFalse($valid);
  }
}
```

PHPUnit

- Running phpunit will now show:



```
[vagrant@vagrant-csy2028 http]$ phpunit
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

.E

Time: 342 ms, Memory: 13.25Mb

There was 1 error:

1) SaveUserTest::testMissingEmail
Undefined index: email

/srv/http/saveuser.php:13
/srv/http/tests/SaveUserTest.php:33

FAILURES!
Tests: 2, Assertions: 1, Errors: 1.
[vagrant@vagrant-csy2028 http]$
```

- Note: PHPUnit also shows PHP errors!

Why automated testing?

- Automated testing allows you to describe how a function *should* work
- The tests contain information about:
  - What inputs are possible
  - What the outputs should be for those inputs

Why automated testing?

- For example, you may write a function that adds together all the elements in an array

```php
<?php

  function addArray($array) {
    $total = $array[0] + $array[1] + $array[2];

    return $total;
  }
```

Testing

- Without automated testing you would just use the function on a real web page and hope it worked
- You probably wouldn't try many variations of how it could be used
- Automated tests allow you to quickly test different criteria to see that the function produces the expected result

Testing

- When calling the function these are the expected results:

```php
<?php
$array = [1, 2, 3];
echo addArray($array); //Should print 6

$array = [2, 2, 1];
echo addArray($array); //Should print 5


$array = [10, 20, 2];
echo addArray($array); //Should print 32

$array = [1, 2, 5, 2];
echo addArray($array); //Should print 10


$array = [10, 20];
echo addArray($array); //Should print 30

$array = [10];
echo addArray($array); //Should print 10

$array = [];
echo addArray($array); //Should print 0
```

PHPUnit

- PHPUnit can be used to check whether or not the expected result matches the actual result
- This will test to see if the function is working as intended.
- Tests can be carried out for all kinds of different inputs to check the function works as it should

PHPUnit

- There are several *assertions* that PHPUnit can issue to check whether a value is as it's expected
- assertEquals can be used to check whether one value is equal to another. E.g to check if $variable is equal to 4:.

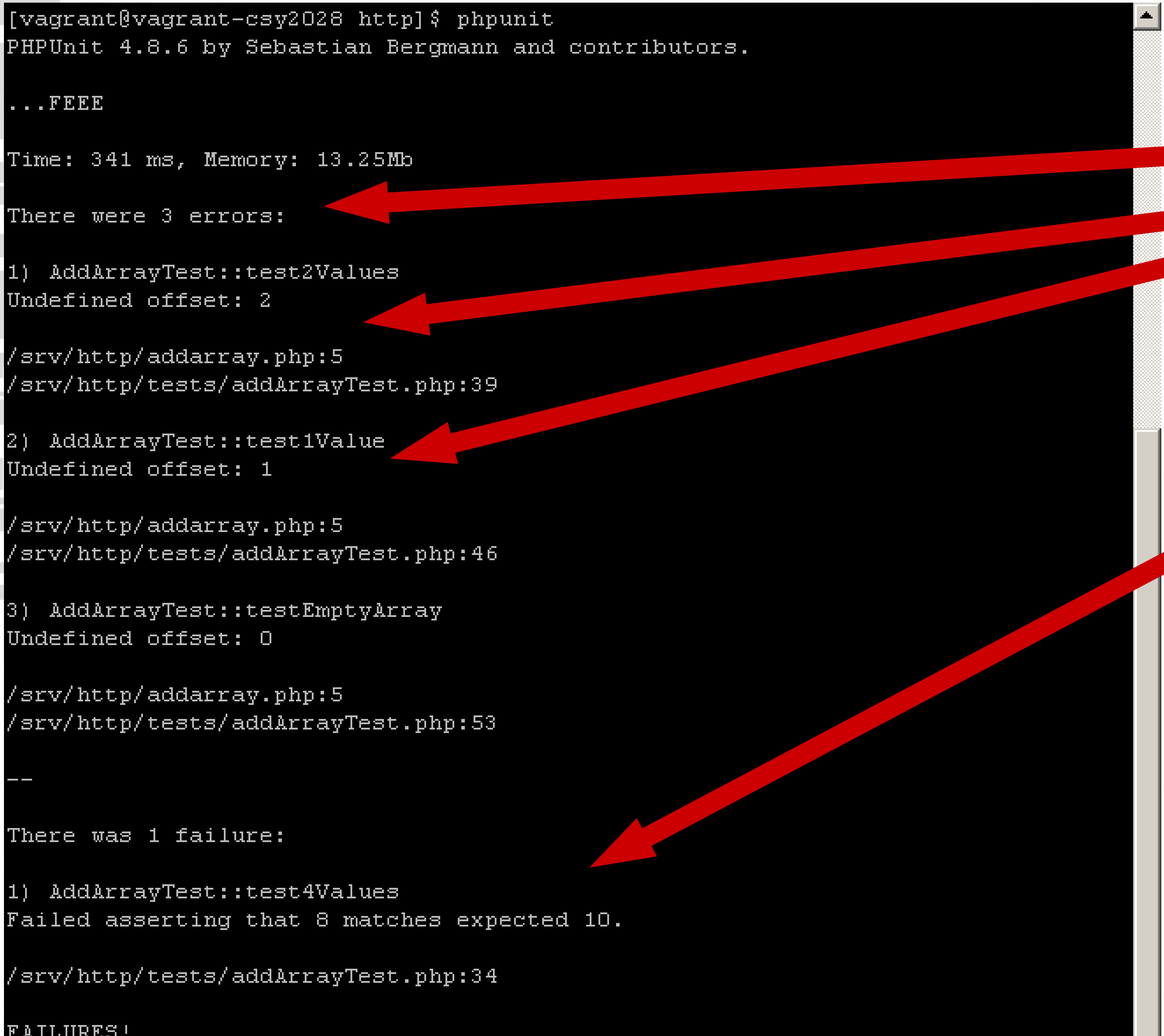
```
    $this->assertEquals($variable, 4);
```

PHPUnit

- This can be used to test the function addArray function works as intended

```php
<?php
require 'addarray.php';
class AddArrayTest extends \PHPUnit\Framework\TestCase {

        public function test3Values() {
                $array = [1, 2, 3];
                $total = addArray($array);

                $this->assertEquals(6, $total);
        }

        public function test3Values2() {
                $array = [2, 2, 1];
                $total = addArray($array);

                $this->assertEquals(5, $total);
        }

        public function test3Values3() {
                $array = [10, 20, 2];
                $total = addArray($array);

                $this->assertEquals(32, $total);
        }
```

```php
        public function test4Values() {
                $array = [1, 2, 5, 2];
                $total = addArray($array);

                $this->assertEquals(10, $total);
        }

        public function test2Values() {
                $array = [10, 20];
                $total = addArray($array);

                $this->assertEquals(30, $total);
        }

        public function test1Value() {
                $array = [10];
                $total = addArray($array);

                $this->assertEquals(10, $total);
        }

        public function testEmptyArray() {
                $array = [];
                $total = addArray($array);

                $this->assertEquals(0, $total);
        }
}
```



```
[vagrant@vagrant-csy2028 http]$ phpunit
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

...FEEE

Time: 341 ms, Memory: 13.25Mb

There were 3 errors:

1) AddArrayTest::test2Values
Undefined offset: 2

/srv/http/addarray.php:5
/srv/http/tests/addArrayTest.php:39

2) AddArrayTest::test1Value
Undefined offset: 1

/srv/http/addarray.php:5
/srv/http/tests/addArrayTest.php:46

3) AddArrayTest::testEmptyArray
Undefined offset: 0

/srv/http/addarray.php:5
/srv/http/tests/addArrayTest.php:53

--

There was 1 failure:

1) AddArrayTest::test4Values
Failed asserting that 8 matches expected 10.

/srv/http/tests/addArrayTest.php:34

FAILURES!
```

"There were 3 errors"
This refers to PHP errors which are listed below with file names and line numbers

Failed tests are also shown

PHPUnit

- The tests describe how the function *should* work

```php
<?php
require 'addarray.php';
class AddArrayTest extends \PHPUnit\Framework\TestCase {


        public function test3Values() {
                $array = [1, 2, 3];
                $total = addArray($array);

                $this->assertEquals(6, $total);
        }

        public function test3Values2() {
                $array = [2, 2, 1];
                $total = addArray($array);

                $this->assertEquals(5, $total);
        }

        public function test3Values3() {
                $array = [10, 20, 2];
                $total = addArray($array);

                $this->assertEquals(32, $total);
        }
```

```php
        public function test4Values() {
                $array = [1, 2, 5, 2];
                $total = addArray($array);

                $this->assertEquals(10, $total);
        }

        public function test2Values() {
                $array = [10, 20];
                $total = addArray($array);

                $this->assertEquals(30, $total);
        }

        public function test1Value() {
                $array = [10];
                $total = addArray($array);

                $this->assertEquals(10, $total);
        }

        public function testEmptyArray() {
                $array = [];
                $total = addArray($array);

                $this->assertEquals(0, $total);
        }
}
```

PHPUnit

- Because the tests describe the way the function *should* work if there are any errors there are bugs in the function they are detected
- The original function looked like this:

```
function addArray($array) {
  $total = $array[0] + $array[1] + $array[2];

  return $total;
}
```

- This works for some cases (Where there are 3 elements in the array) but not for all cases. Some tests pass, some fail

PHPUnit

- This means there is a bug somewhere in the code
- The code always adds the first 3 array elements:
- Any elements after the first 3 will not be added
- If there are fewer than 3 elements, it will cause an error

```
<?php
  function addArray($array) {
    $total = $array[0] + $array[1] + $array[2];

    return $total;
  }
```

PHPUnit

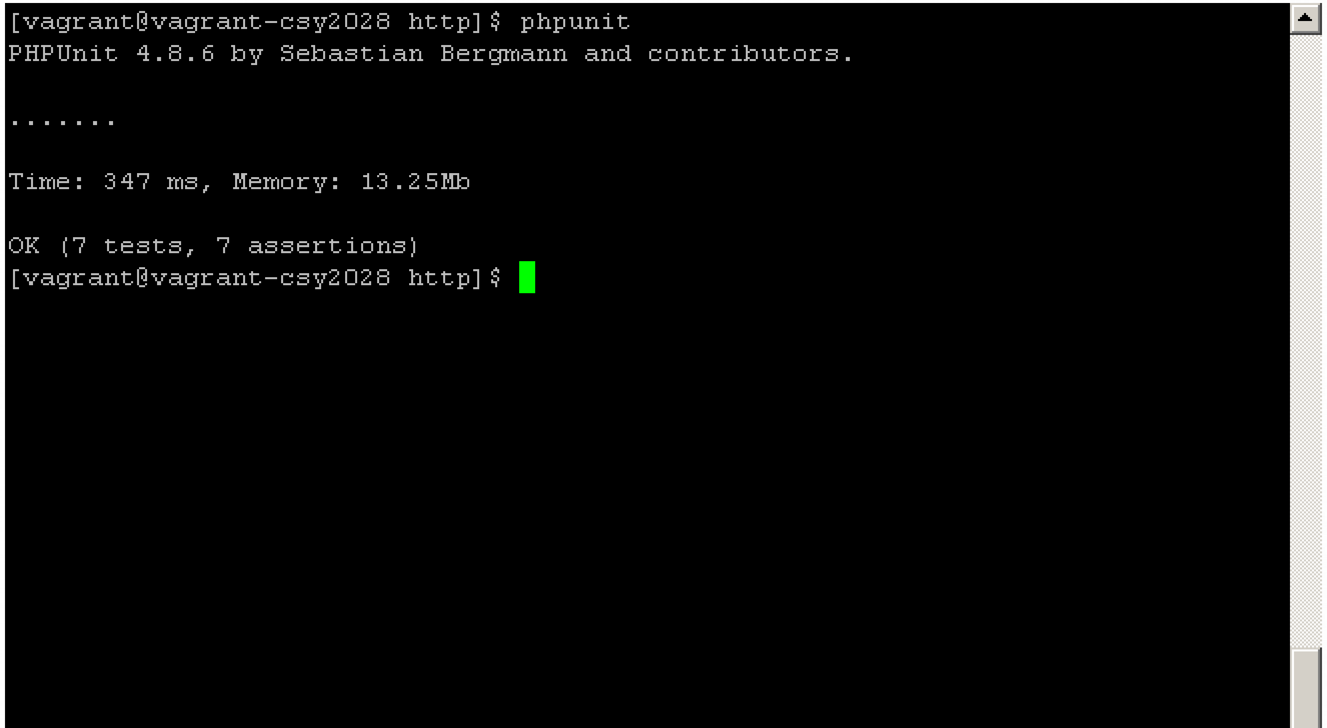
- To fix the function it should take an array of any size. This can be done with a loop:

```
function addArray($array) {
  $total = $array[0] + $array[1] + $array[2];

  return $total;
}
```

```
function addArray($array) {
  $total = 0;

  foreach ($array as $value) {
      $total = $total + $value;
  }

  return $total;
}
```

PHPUnit

- If the bugs have been fixed, all the tests should pass:



```
[vagrant@vagrant-csy2028 http]$ phpunit
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

.......

Time: 347 ms, Memory: 13.25Mb

OK (7 tests, 7 assertions)
[vagrant@vagrant-csy2028 http]$
```

Exercise 1

- Download ex1.zip
- This contains a set of tests for a function called ArrayCount which counts the number of times an element appears in an array
- The arrayCount function should take two arguments:
  - 1) The array
  - 2) The item to search for
- ArrayCount.php gives you some examples (and tests) for how the function should work.
- Write the function so that the tests pass

Welcome to Test Driven Development (TDD)

- Test-Driven-Development (TDD) is a development methodology that is heavily reliant on automated testing
- In TDD tests are written *before* any code
- The tests are written to describe how a function or class *should work*
- To start with, all tests will fail
- The code is then written so that tests will pass

Test Driven Development (TDD)

- Tests are run frequently during development:
  - Code is written until one test passes
  - All tests are run
  - Code is written for the next test
  - All tests are run
  - If any changes to the code stop other tests passing it's immediately obvious a bug has been introduced

TDD

- Under TDD every line of code should be tested
- PHPUnit contains a *code coverage report* which can highlight which lines of code have been tested and which haven’t

TDD Example

- Consider the following function:

```
function arrayContains($array, $value) {
        $found = false;

        for ($i = 0; $i < count($array); $i++) {
                if ($array[$i] == $value) {
                        $found = true;
                }
        }

        return $found;
}
```

- This searches an array and returns true or false if the $value supplied is contained within the array $array

```
$array = [1, 3, 4];

$result =  arrayContains($array, 2);
//$result should be false


$result =  arrayContains($array, 3);
//$result should be true
```

Testing

- A test could be added for this code

```
<?php
require 'includes/arraycontains.php';
class ArrayContainsTest extends \PHPUnit\Framework\TestCase {

}
```

- This test could be run but there are no tests
- However, it’s possible to generate a coverage report to see what has been tested

Code coverage

- To do this you need to tell PHPUnit which files to include in the code coverage report
- This is done using the phpunit.xml configuration file. Tell PHP which files you are testing

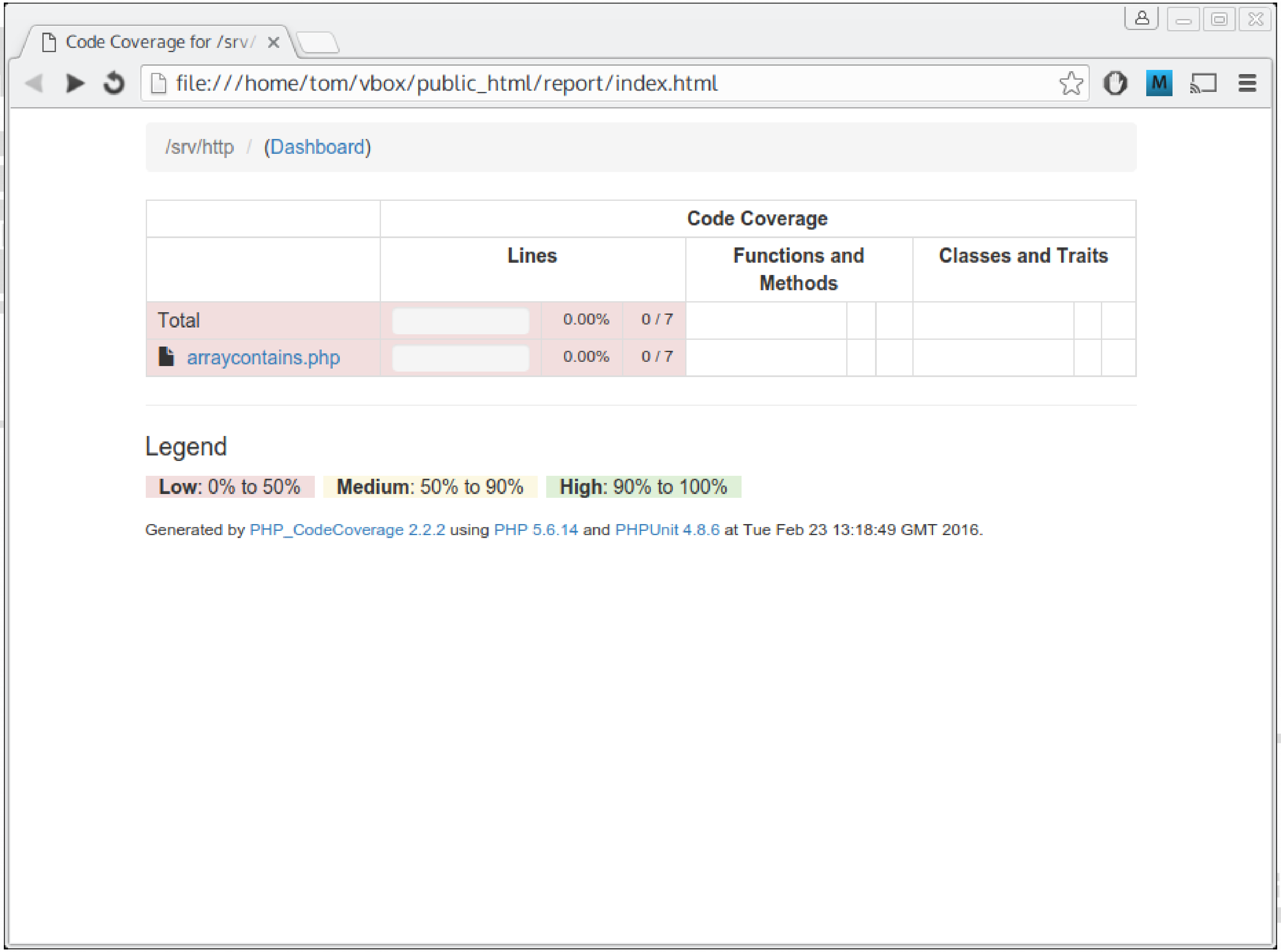
```
<?xml version="1.0"?>
<phpunit>
  <testsuites>
    <testsuite name="tests">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
 <filter>
  <whitelist processUncoveredFilesFromWhitelist="true">
    <file>includes/arraycontains.php</file>
  </whitelist>
</filter>
</phpunit>
```

Code coverage

- However, it's possible get PHPUnit to generate a *code coverage report* which will show which parts of the code have been tested
- This is done by running the phpunit command with an argumnet:
- `phpunit --coverage-html=./report`

Code coverage

- phpunit --coverage-html=./report
- This will generate a set of HTML files in the directory /report
- If you open up the file index.html in your browser you see the code coverage report:



/srv/http / (Dashboard)

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		0.00%	0 / 7						
arraycontains.php		0.00%	0 / 7						

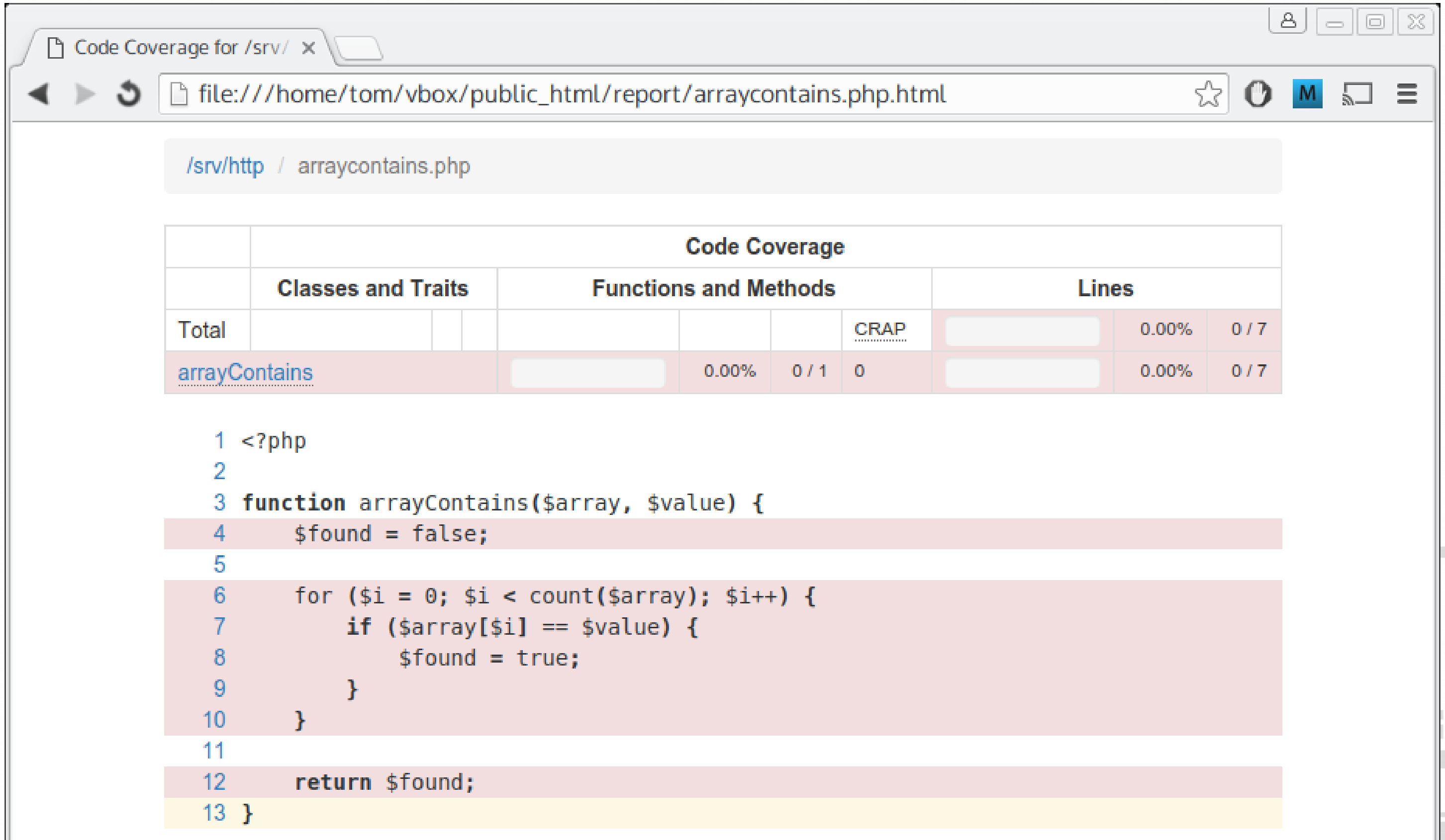
Legend

Low: 0% to 50% **Medium**: 50% to 90% **High**: 90% to 100%

Generated by PHP_CodeCoverage 2.2.2 using PHP 5.6.14 and PHPUnit 4.8.6 at Tue Feb 23 13:18:49 GMT 2016.

Coverage report

- This lists all the files which are being tested, if you click on one you can see which lines of code have been tested:



Code Coverage for /srv/

file:///home/tom/vbox/public_html/report/arraycontains.php.html

/srv/http / arraycontains.php

	Code Coverage									
	Classes and Traits			Functions and Methods				Lines		
Total							CRAP		0.00%	0 / 7
arrayContains					0.00%	0 / 1	0		0.00%	0 / 7

```php
<?php

function arrayContains($array, $value) {
    $found = false;

    for ($i = 0; $i < count($array); $i++) {
        if ($array[$i] == $value) {
            $found = true;
        }
    }

    return $found;
}
```

TDD

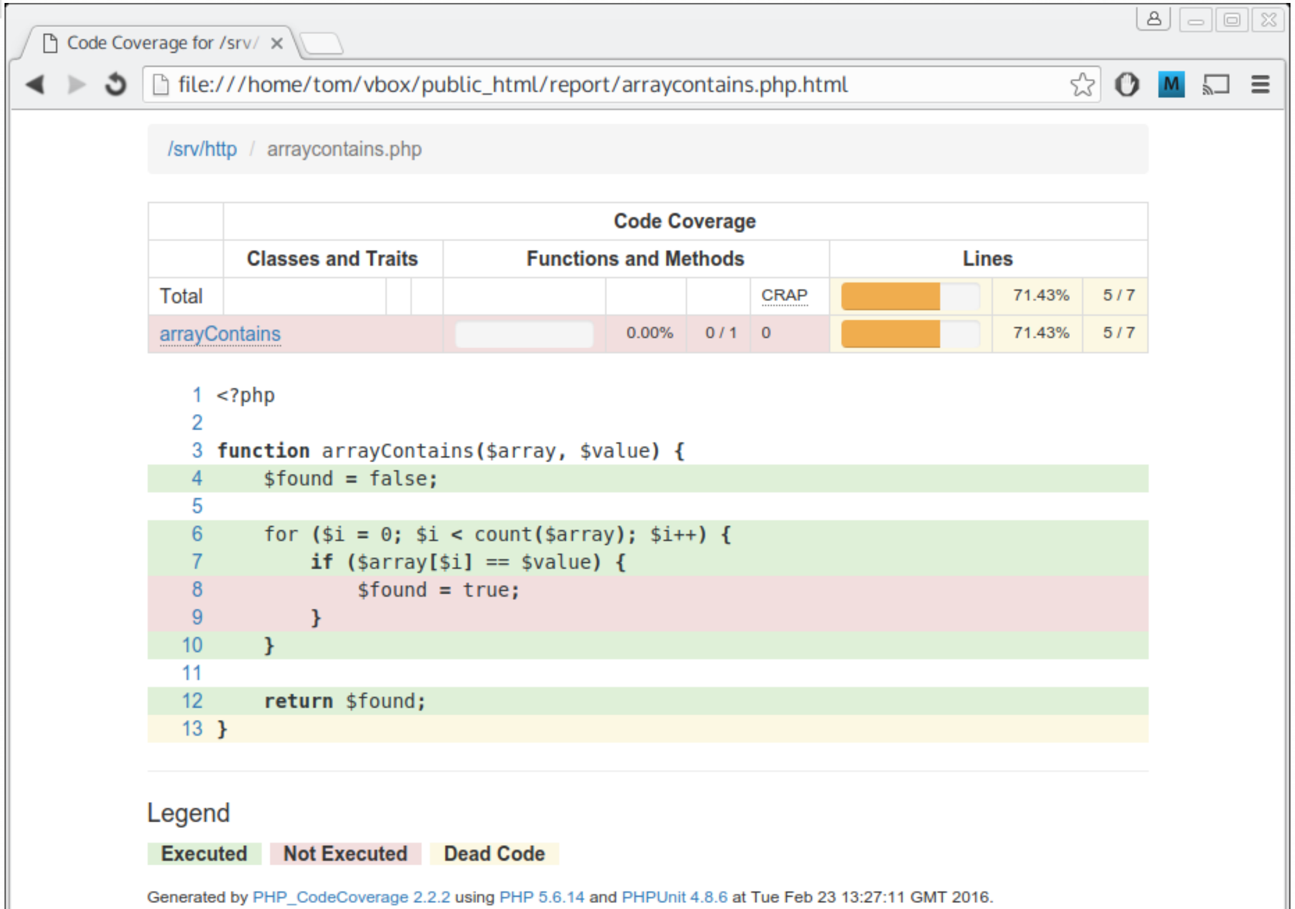
- If a test is added for the “false” condition in the arraycontains function

```
$array = [1, 3, 4];

$result =  arrayContains($array, 2);
//$result should be false


$result =  arrayContains($array, 3);
//$result should be true
```

```
require_once 'arraycontains.php';

class ArrayContainsTest extends \PHPUnit\Framework\TestCase {

        public function testDoesNotContain() {
                $array = [1, 3, 5];

                //Result should be false as $array does not contain 4
                $result = arrayContains($array, 4);

                $this->assertFalse($result);
        }
}
```

Code coverage

- If PHPUnit is run again with the coverage report it now shows which lines have been tested:



Code Coverage for /srv/

file:///home/tom/vbox/public_html/report/arraycontains.php.html

/srv/http / arraycontains.php

	Code Coverage									
	Classes and Traits			Functions and Methods				Lines		
Total							CRAP		71.43%	5 / 7
arrayContains					0.00%	0 / 1	0		71.43%	5 / 7

```php
<?php

function arrayContains($array, $value) {
    $found = false;

    for ($i = 0; $i < count($array); $i++) {
        if ($array[$i] == $value) {
            $found = true;
        }
    }

    return $found;
}
```

Legend

Executed Not Executed Dead Code

Generated by PHP_CodeCoverage 2.2.2 using PHP 5.6.14 and PHPUnit 4.8.6 at Tue Feb 23 13:27:11 GMT 2016.

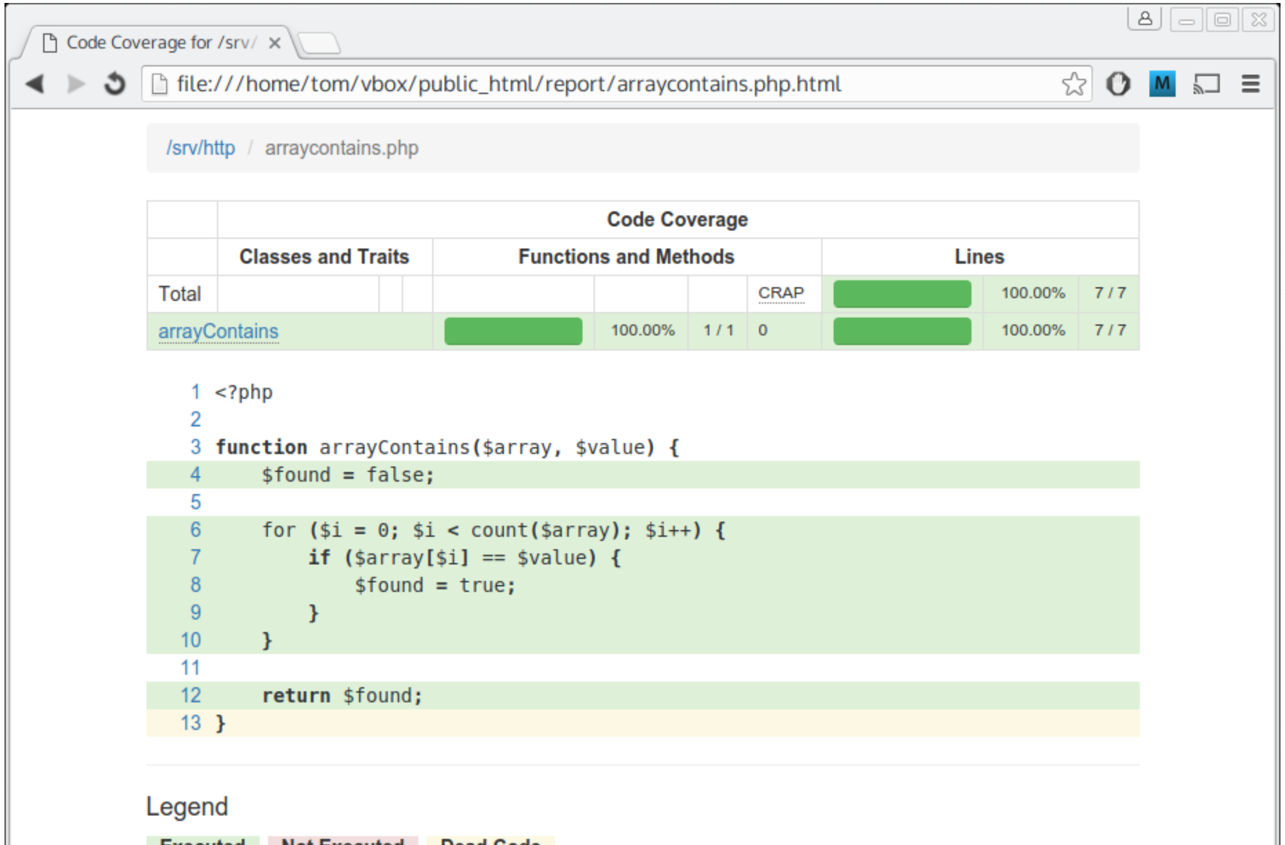
Code coverage

- Ideally, all possible outcomes of the function should be tested, this requires 100% code coverage
- To do this another test can be added to test the true condition

```php
require_once 'arraycontains.php';

class ArrayContainsTest extends \PHPUnit\Framework\TestCase {

        public function testDoesNotContain() {
                $array = [1, 3, 5];

                //Result should be false as $array does not contain 4
                $result = arrayContains($array, 4);

                $this->assertFalse($result);
        }

        public function testDoesContain() {
                $array = [1, 3, 5];

                //Result should be true as $array does contain 5
                $result = arrayContains($array, 5);

                $this->assertTrue($result);
        }
}
```

Code coverage

- The coverage report now shows 100%. This means every possible outcome of the function has been tested


Code Coverage for /srv/

file:///home/tom/vbox/public_html/report/arraycontains.php.html

/srv/http / arraycontains.php

	Code Coverage									
	Classes and Traits			Functions and Methods				Lines		
Total							CRAP	🟩	100.00%	7 / 7
arrayContains				🟩	100.00%	1 / 1	0	🟩	100.00%	7 / 7

```
<?php

function arrayContains($array, $value) {
    $found = false;

    for ($i = 0; $i < count($array); $i++) {
        if ($array[$i] == $value) {
            $found = true;
        }
    }

    return $found;
}
```

Legend

Testing the save function

- Last week ended by testing the saveUser function

```
function saveUser($users, $user) {
    $valid = false;

    if ($user['firstname'] == '') {
        $valid = false;
    }

    if ($user['surname'] == '') {
        $valid = false;
    }

    if ($user['email'] == '') {
        $valid = false;
    }

    if ($valid) {
        $users->save($user);
    }

    return $valid;
```

Automated Testing

- The test for success looks like this:

```php
<?php

require 'saveuser.php';
require 'databasetable.php';

class SaveUserTest extends \PHPUnit\Framework\TestCase {
  public function testValidData() {
    $pdo = new PDO('mysql:host=192.168.56.2;dbname=csy2028', 'student', 'student');
    $users = new DatabaseTable($pdo, 'user');

    $validData = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org'
    ];


    $valid = saveUser($users, $validData);


    $this->assertTrue($valid);

    $stmt = $pdo->query('SELECT * FROM user WHERE email = :email');
    $criteria = [
        'email' => 'john@example.org'
    ];

    $stmt->execute($criteria);

    $record = $stmt->fetch();
    $this->assertNotFalse($record);
    $this->assertEquals($record['firstname'], $validData['firstname']);
    $this->assertEquals($record['surname'], $validData['surname']);
    $this->assertEquals($record['email'], $validData['email']);


  }
}
```

Run the function with valid data

Check that the validation works

This only checks whether the validation
Check is happening correctly

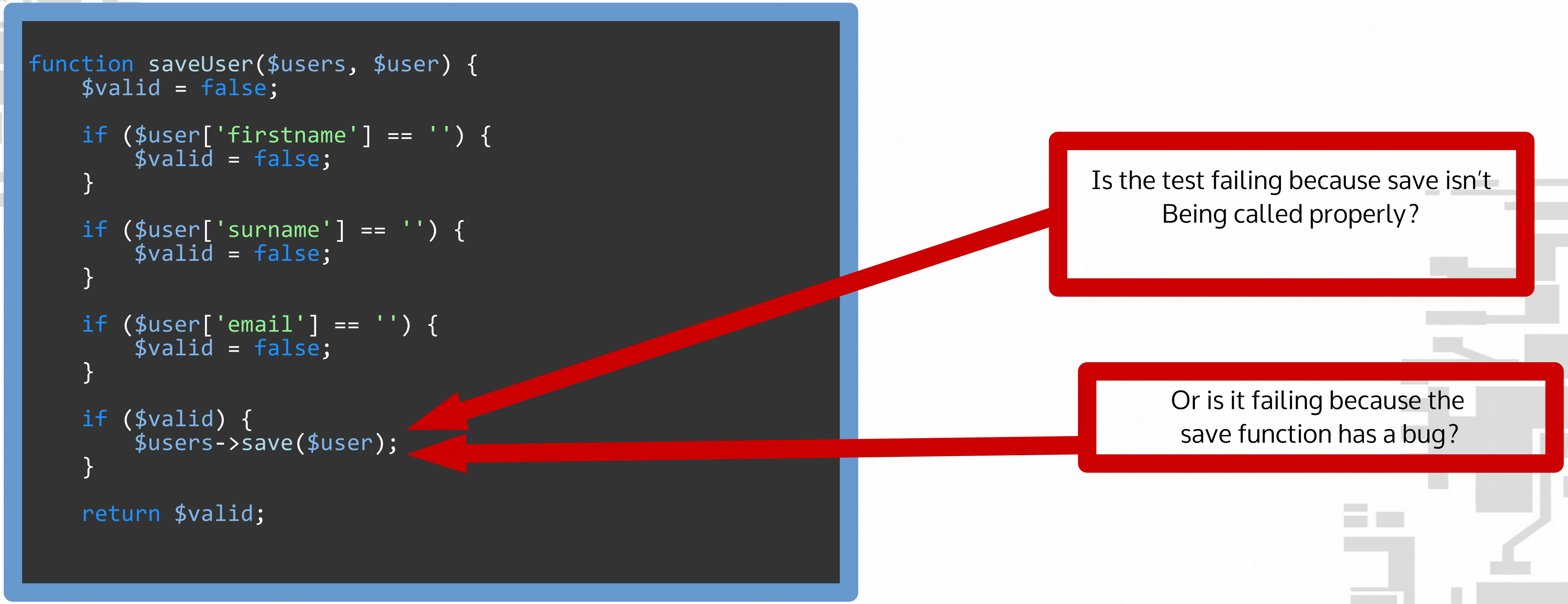
Query the database for the record
that was just inserted

If no record was found, the function
isn't working correctly

Finally check each field
has been set to the correct
value

Testing

- This is actually testing more than the the function. It's also testing the DatabaseTable class
- However, if the test fails, is it easy to track down the problem?



```
function saveUser($users, $user) {
    $valid = false;

    if ($user['firstname'] == '') {
        $valid = false;
    }

    if ($user['surname'] == '') {
        $valid = false;
    }

    if ($user['email'] == '') {
        $valid = false;
    }

    if ($valid) {
        $users->save($user);
    }

    return $valid;
```

Is the test failing because save isn't Being called properly?
Or is it failing because the save function has a bug?

Isolation

- Ideally to test the saveUser function, only the code in that function should be tested. The previous test is actually checking both the saveUser() function **and** the code in the save() function in the DatabaseTable class.
- If the record doesn't insert for some reason it's not possible to known exactly where the problem is

Mock Objects

- It's better to remove any function calls from the $user object so that only code in the saveUser function is run
- Otherwise to test the function
  - the object needs to be created
  - A real database connection is required
  - All the constructor arguments for the DatabaseTable class need to be supplied to run the test

Mock Objects

- PHPUnit allows you to create fake “mock” objects automatically
- This is like creating a different DatabaseTable class that actually has no functionality:

```
class MockDatabaseTable {
    public function save($data) {

    }
}
```

Mock Objects

- This would allow the test to be run:
  - Without a real database connection
  - Without a real DatabaseTable instance
- If the test fails it fails because of code in the function being tested, not because of code in the DatabaseTable class

- By using a fake ("mock") database table class, it's possible to run the function without a real database connection and without running any other code

```php
<?php

require 'saveuser.php';

class SaveUserTest extends \PHPUnit\Framework\TestCase {
  public function testValidData() {
    $users = new MockDatabaseTable();

    $validData = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org'
    ];


    $valid = saveUser($users, $validData);


    $this->assertTrue($valid);

  }
}
```

Mock Objects

- The problem with this test is that it never checks that the DatabaseTable's save function has been run with the correct parameters!
- This could be done by adding a check to the mock class:

```
class DatabaseTable {
    public $validData = true;

    public function save($data) {
        if ($data['firstname'] == 'John'
            && $data['surnamne'] == 'Smith'
            && $data['email'] == 'john@example.org') {
            $this->valid = true;
        }
    }
}
```

Mock Objects

- An assertion can be made to check whether the correct data was sent to the save() function

```php
<?php

require 'saveuser.php';
require 'mockdatabase.php';

class SaveUserTest extends \PHPUnit\Framework\TestCase {
  public function testValidData() {
    $users = new MockDatabaseTable();

    $validData = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org'
    ];


    $valid = saveUser($users, $validData);


    $this->assertTrue($valid);
    $this->assertTrue($users->valid);

  }
}
```

```php
function saveUser($users, $user) {
    $valid = false;

    if ($user['firstname'] == '') {
        $valid = false;
    }

    if ($user['surname'] == '') {
        $valid = false;
    }

    if ($user['email'] == '') {
        $valid = false;
    }

    if ($valid) {
        $users->save($user);
    }

    return $valid;
}
```

```php
class MockDatabaseTable {
    public $validData = true;

    public function save($data) {
        if ($data['firstname'] == 'John'
            && $data['surnamne'] == 'Smith'
            && $data['email'] == 'john@example.org') {
            $this->valid = true;
        }
    }
}
```

Mock Objects

- This checks both whether the validation has passed and that the save function is called with the correct data
  - Without testing the actual save function!
  - A separate test could be done to check the save() function works as intended
- Now, if the test failes it must be due to a problem in the saveUser() function and you know exactly where to look

PHPUnit Mocks

- Creating a mock class for each test can be time consuming
- PHPUnit allows you to quickly create a mock object without needing to write a class
- This is done using:

```
    $this->getMockBuilder('SomeClass')->getMock();
```

- Mocks can be generated from any class

PHPUnit Mocks

- You can specify that you are expecting a specific function to be called on on the class with specified parameters
- If the function is not called with those parameters the test will fail
- This is done using:

```
$mock = $this->getMockBuilder('SomeClass')->getMock();
  $mock->expects($this->once())
       ->method('NameOfMethod')
       ->with($this->equalTo('Expected argument to method'));
```

PHPUnit Mocks

- Once the mock is created it can be used in place of the original object
- E.g. for the saveUser function:

```php
<?php
require 'saveuser.php';

class SaveUserTest extends \PHPUnit\Framework\TestCase {
  public function testValidData() {


    $validData = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org'
    ];

    $users = $this->getMockBuilder('DatabaseTable')->getMock();
    $users->expects($this->once())
          ->method('save')
          ->with($this->equalTo($validData));

    $valid = saveUser($users, $validData);


    $this->assertTrue($valid);
  }
}
```

Making mock objects return values

- You can use the `WillReturn` function to set a return value when the functon is called with those particular arguments

```
$mock = $this->getMockBuilder('SomeClass')->getMock();
  $mock->expects($this->once())
       ->method('NameOfMethod')
       ->with($this->equalTo('Expected argument to method'))
       ->willReturn('value the mock should return');
```

PHPUnit

- PHPUnit can do a lot more with mocks and assertions. For more information and examples, see the PHPUnit website: https://phpunit.de/

Writing testable code

- The way you write and structure your code has an impact on how easy code is to isolate and test

- Which of these is easier to test?

```php
function saveUser($users, $user) {
    $valid = false;

    if ($user['firstname'] == '') {
        $valid = false;
    }

    if ($user['surname'] == '') {
        $valid = false;
    }

    if ($user['email'] == '') {
        $valid = false;
    }

    if ($valid) {
        $users->save($user);
    }

    return $valid;
}
```

```php
function saveUser($pdo, $user) {
    $valid = false;

    if ($user['firstname'] == '') {
        $valid = false;
    }

    if ($user['surname'] == '') {
        $valid = false;
    }

    if ($user['email'] == '') {
        $valid = false;
    }

    if ($valid) {
        $users = new DatabaseTable($pdo,'users');
        $users->save($user);
    }

    return $valid;
}
```

Exercise 2

- Download ex2.zip and extract it
- The supplied arrayCountAll function takes an array and counts the number of times any value as found, returning an array e.g. the input
  - ['A', 'A', 'A', 'B']
  - Will generate the output ['A' => 3, 'B' => 1]
- Write tests to check that the function works as intended
- Generate a code coverage report to ensure that all lines have been tested

Exercise 3

- Write a function arrayReverse() which takes an array and reverses the order of elements.
- Create the relevant tests to ensure the function works as intended and generate a code coverage report. Ensure you get 100% code coverage

Exercise 4

- Write a function that takes a DatabaseTable class and generates a HTML table from it.
- For example
-

```
  function generateTable($databaseTable) {
    //...

  }


$users = new DatabaseTable($valid, 'users');
echo generateTable($users);
```

```
<table>
        <thead>
                <th>Firstname</th>
                <th>Surname</th>
                <th>Email</th>
        </thead>
        <tbody>
                <tr>
                        <td>John</td>
                        <td>Smith</td>
                        <td>john@example.org</td>
                </tr>
                <tr>
                        <td>Sue</td>
                        <td>Evans</td>
                        <td>sue@example.org</td>
                </tr>
        </tbody>
</table>
```

Exercise 4

- Write tests to check the function works as epected by using Mock Objects in place of the `DatabastTable` class