CSY2028
Web Programming
Topic 15

Tom Butler
thomas.butler@northampton.ac.uk

Quick note

- You may need to run the following commands for today's lecture:
  - vagrant box update
  - vagrant destroy
  - vagrant up

Topic 15

- Writing testable code
- Unit testing with PHPUnit

Processing forms

- Last week we separated out the HTML logic from the database logic and ended up with code like this:

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

//Include the file that contains the DatabaseTable class and the $pdo object
require 'database.php';

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

if (isset($_POST['submit'])) {
        $users->save($_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);
```

Form Validation

- Let's add some validation to the form:
  - Data is only inserted into the database if it is valid

```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'];
}
```

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 (filling in other fields)
    - Checking the error is shown
  - Leaving surname blank (filling in other fields)
    - Checking the error is shown
  - Leaving email blank (filling in other fields)
    - Checking the error is shown
  - Leaving two of the three fields blank (filling in other fields)
    - 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!
- You should ideally run all tests again to ensure no new bugs have been introduced. For this page that means running all 6 tests again
- If another bug is found, the process needs to be repeate
- And repeated until all tests are successful **and there are no bugs remaining**

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

Programmers are lazy

- Testing can be boring
- When a bug is fixed, it's easy to ignore the fact that the fix might have introduced a new bug
- Ideally you should run all the other tests again each time you make a change
- This is time consuming and often a programmer will assume that nothing will be broken
- Often that assumption is wrong!

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;
    }

//...
```

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
  - Not that the template is displayed as expected
  - Or that it's inserted into the HTML layout

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;
```

Automated testing

- Once the function has been separated, it's possible to run it from anywhere
- One such place is code that is used to test the function is producing the correct outcome

```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;
```

Automated Testing

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

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


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

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


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

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

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


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

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

Automated Testing

- And to test the complete function it's also possible to test to see whether a record has been added to the database correctly:

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

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

//This is supposed to be valid
if ($valid == true) {
   $pdo->prepare('SELECT * FROM user WHERE email = :email');


   $stmt = $pdo->execute(['email' => 'john@example.org']);

   //Retrieve the record back out of the database
   $record = $stmt->fetch();

   if ($record == false) {
      echo 'Test failed';
   }
   else {
       if ($record['firstname'] == $validRecord['firstname']
        && $record['surname'] == $validRecord['surname']
        && $record['email'] == $validRecord['email']) {
            echo 'Test passed';
       }
       else {
         echo 'Test failed';
       }
   }
}
else {
    echo 'Test failed';
}
```

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

Exercise 1

- Write some test code for the following function to locate bugs

```php
function isPass($percentage) {
    if ($percenage < 40) {
        return true;
    }
    else {
        return false;
    }
}
```

- The function takes a percentage as a number and returns true/false depending on whether the percentage is over 40. Pass grades are 40+ marks
- Place the function in its own file (isPass.php) and in another file (testIsPass.php)write code to test the function works as intended and fix any bugs you find

Exercise 1 - Solution

```php
<?php

require 'isPass.php';


if (isPass(20) == false) {
        echo 'Test passed';
}
else {
        echo 'Test failed - 20 should be false';
}

if (isPass(50) == true) {
        echo 'Test passed';
}
else {
        echo 'Test failed - 50 should be false';
}

if (isPass(40) == true) {
        echo 'Test passed';
}
else {
        echo 'Test failed - 40 should be true';
}
```

```php
function isPass($percentage) {
    if ($percentage >= 40) {
        return true;
    }
    else {
        return false;
    }
}
```

Automated testing

- Now, after making any changes to the function, we can just run testIsPass.php to check no new bugs have been introduced
- We can test the function works in isolation, we don't need to use it as part of a process on a real website. We can call the function on our website and know it will work

```php
<?php

require 'isPass.php';


if (isPass(20) == false) {
        echo 'Test passed';
}
else {
        echo 'Test failed - 20 should be false';
}

if (isPass(50) == true) {
        echo 'Test passed';
}
else {
        echo 'Test failed - 50 should be false';
}

if (isPass(40) == true) {
        echo 'Test passed';
}
else {
        echo 'Test failed - 40 should be true';
}
```

Automated Testing

- This testing process can be a lot of code
- In this case, there is more code for the test than the function being tested
- However, once the code has been written, any time you make a change to the function you can quickly run all the tests and make sure it is working correctly

Automated Testing

- By placing all the tests for a function in a single file e.g. testSaveUser.php you can quickly test all the possible outcomes of a the function
- Ideally the test should test every line of code inside the function is working as intended

PHPUnit

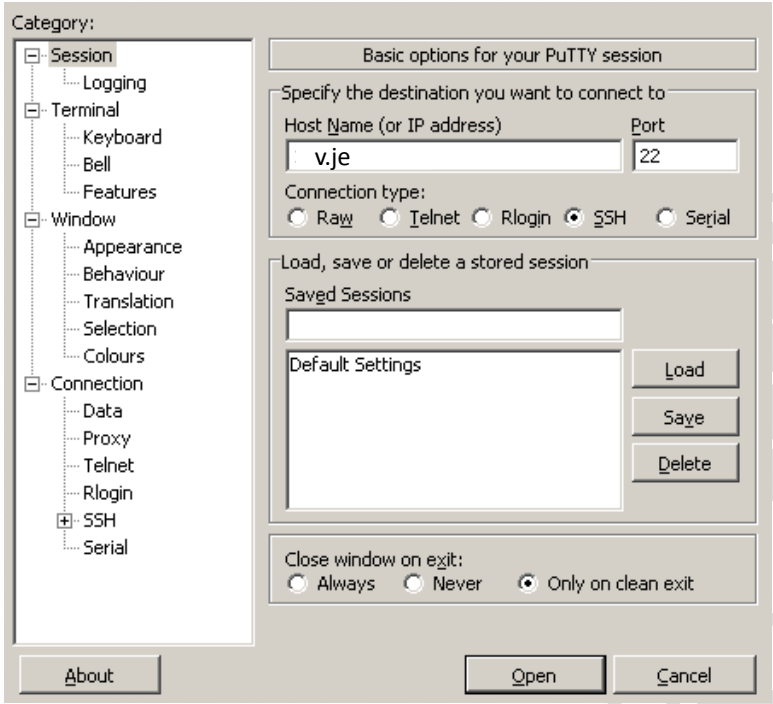
- Automated Testing is a very common problem that has to be solved
- As with most common problems there are existing solutions
- For PHP there is a library called PHPUnit which allows you to quickly write automated tests and includes tools to give you better information about why a test failed

PHPUnit

- To run PHPUnit firstly you must connect to the virtual server
- This can be done via a protocol called SSH. There are many programs which can connect to SSH, one common Windows program is *Putty (http://www.chiark.greenend.org.uk/~sgtatham/putty/)*
- Linux and MacOS come with an SSH utility inbuilt which can be run via command line

Putty

- When you run Putty you will be asked for a server address
- This is the same server address you have been using to connect to the web server and MySQL server: v.je



Linux/Mac

- On Linux or mac open up your command prompt/terminal and type:
  - ssh vagrant@v.je

Putty

- After connecting you might see a security prompt. Type "yes" and hit enter
- Now you will see a log in prompt
- Enter `vagrant` as the username and `vagrant` as the password (Windows)
- Enter `vagrant` as the password (Linux/Mac)
- Note: You will not see the password as you type!
- Once you have entered the username and password you are presented with a command prompt

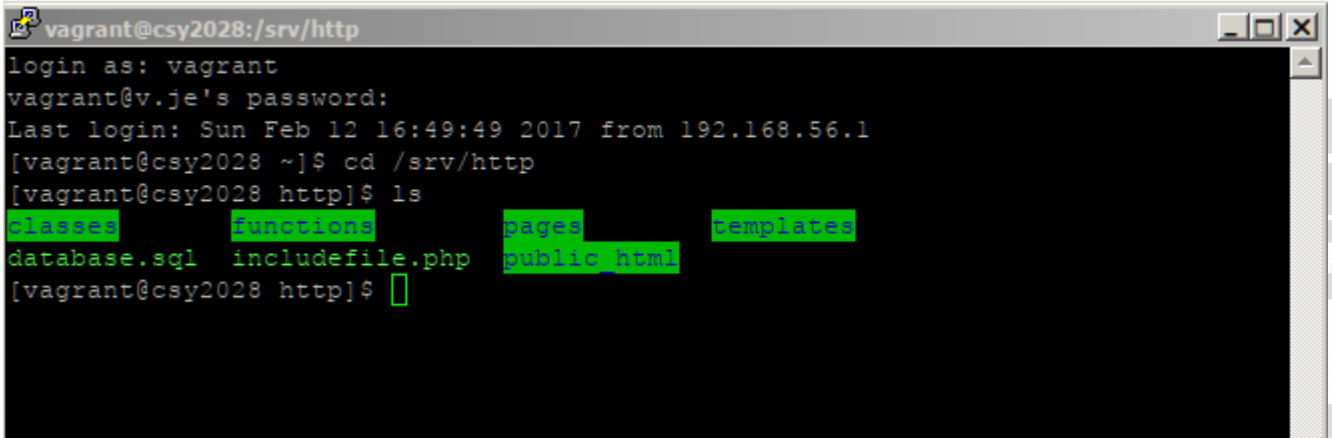
Putty

Command prompt

- There are many possible commands you can type here
- The one we're interested in is `phpunit` however, some configuration needs to be done first
- Firstly `**cd /srv/http**` (exactly! No extra spaces, forward slashes and a space between "cd" and "/srv/http" and press enter. This will navigate to the directory that is linked to your public_html directory

Ls command

- The command `**ls**` is shorthand for "list" and it lists the contents of the current directory
- If you type `**ls**` and press return you should see anything in your `website` directory including your `public_html` directory
- If you don't see a list of files from your `website` directory, check you typed the "cd /srv/http" command correctly.



PHPUnit

- If you run the phpunit command (all lower case) you will see some output:

```
  --loader <loader>          TestSuiteLoader implementation to use.
  --repeat <times>           Runs the test(s) repeatedly.
  --tap                      Report test execution progress in TAP format.
  --testdox                  Report test execution progress in TestDox format.
  --printer <printer>        TestListener implementation to use.

Configuration Options:

  --bootstrap <file>         A "bootstrap" PHP file that is run before the tests.
  -c|--configuration <file> Read configuration from XML file.
  --no-configuration         Ignore default configuration file (phpunit.xml).
  --no-coverage              Ignore code coverage configuration.
  --include-path <path(s)>   Prepend PHP's include_path with given path(s).
  -d key[=value]             Sets a php.ini value.

Miscellaneous Options:

  -h|--help                  Prints this usage information.
  --version                  Prints the version and exits.

  --check-version            Check whether PHPUnit is the latest version.
  --self-update              Update PHPUnit to the latest version.
[vagrant@vagrant-csy2028 http]$
```

PHPUnit Configuration

- Before running any tests with PHPUnit you need to tell it which of your PHP files contains your tests
- This is done using a file called **phpunit.xml** which can be placed inside the **website** directory
- Generally the configuration can look like this:

phpunit.xml

```xml
<?xml version="1.0"?>
<phpunit>
  <testsuites>
    <testsuite name="tests">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

Name of the directory that
Contains your tests
Usually “tests”

In most cases you can just use
this phpunit.xml as is

PHPUnit

- Now if you run the phpunit command you will see a different output:

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



Time: 324 ms, Memory: 12.75Mb

No tests executed!
[vagrant@vagrant-csy2028 http]$
```

PHPUnit

- This means that the configuration file has been loaded and it is scanning the tests directory for suitable files

PHPUnit

- Once you have created the phpunit.xml in the website directory you need to create the *tests* directory to store all your tests
- To create a test you need to create a file that ends in the word Test (With an uppercase T, it is case sensitive!)

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

- If you run PHPUnit you will now see:

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

F

Time: 339 ms, Memory: 13.25Mb

There was 1 failure:

1) Warning
No tests found in class "SaveUserTest".

FAILURES!
Tests: 1, Assertions: 0, Failures: 1.
[vagrant@vagrant-csy2028 http]$
```

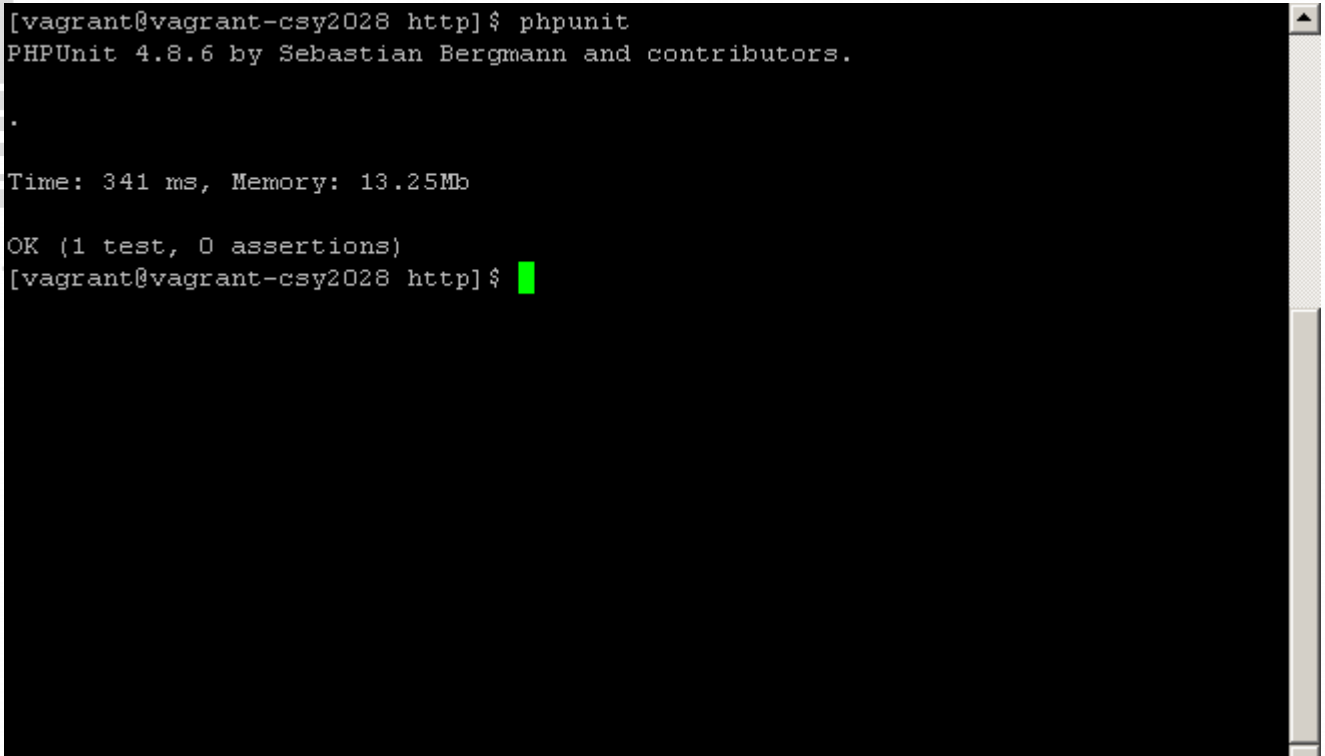
A Basic test

- It says “No tests found” currently nothing is being tested
- Each test is a function inside the class
- To create a test add a public function with a name starting `test` e.g.

```php
<?php

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

  }
}
```

A Basic test

- Running phpunit again will now produce the output:



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

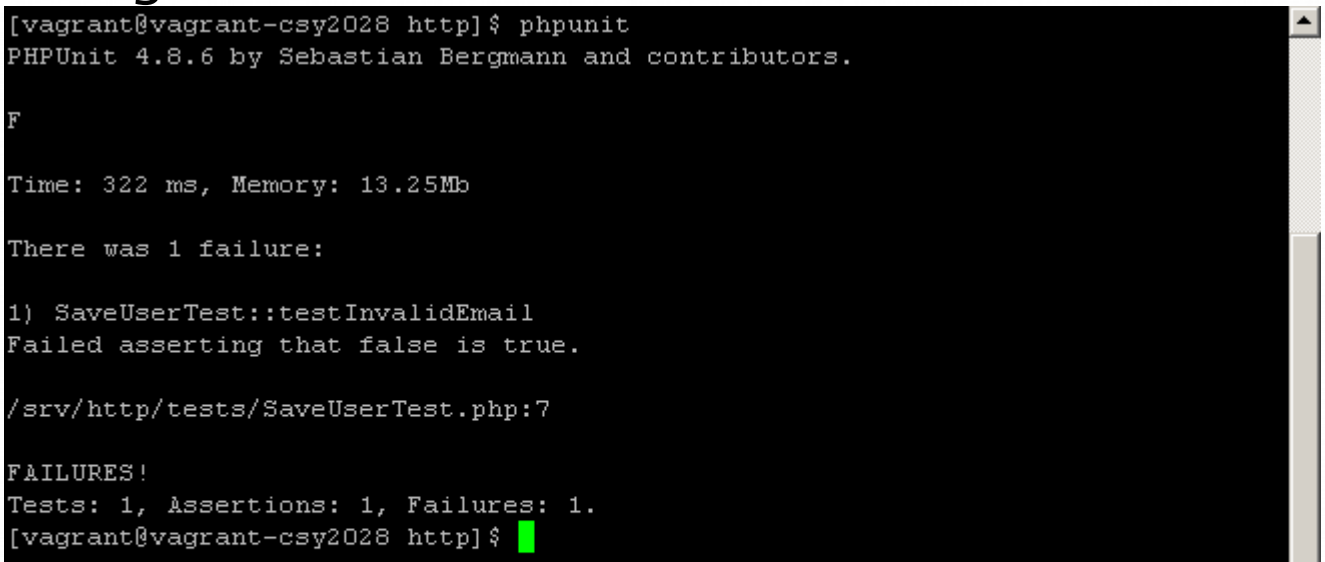
- To test a value is true you can use this:

```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)



A real test

- To test a real function the file that includes the function must be included (The path is relative to the phpunit.xml file so)
- E.g. if the saveUser function is stored in saveuser.php
- You will also need to include any files that are required to run the function (E.g. database connections and other classes)
- To run the test you will need to create all the objects and classes that are required to run the function (More on this next week!)

```php
<?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);
  }
}
```

```php
<?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:

```php
<?php

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

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

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


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

    $this->assertFalse($valid);
  }

  public function testMissingEmail() {
    $pdo = new PDO('mysql:host=v.je;dbname=csy2028', 'student', 'student');
    $users = new DatabaseTable($pdo, 'user');

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

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

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

```php
<?php

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

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

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


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

    $this->assertFalse($valid);
  }

  public function testMissingEmail() {
    $pdo = new PDO('mysql:host=v.je;dbname=csy2028', 'student', 'student');
    $users = new DatabaseTable($pdo, 'user');

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

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

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

Note the duplicated code

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!

PHPUnit

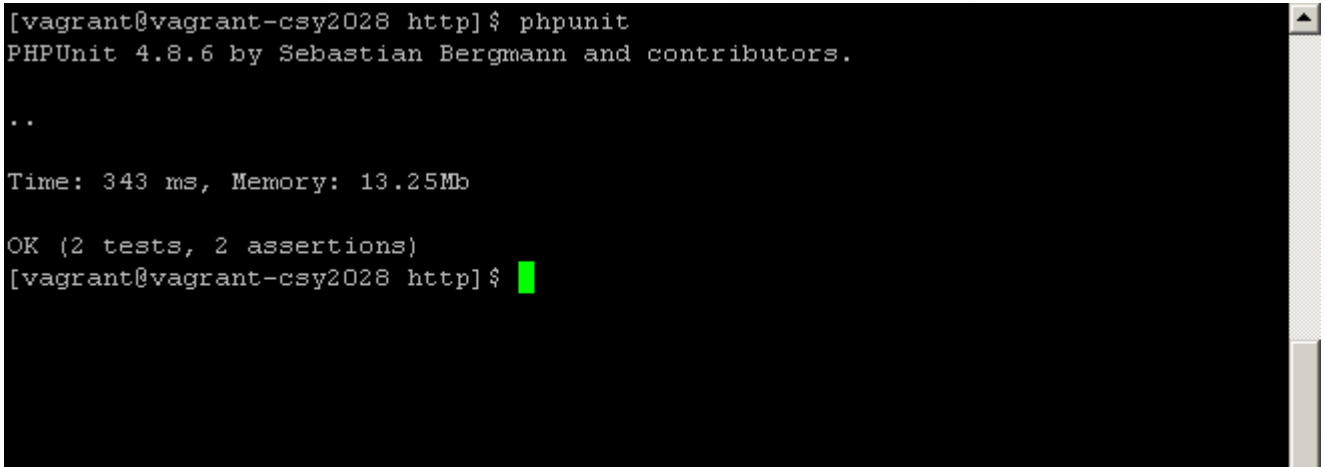
- The error occurs because there is no check to see if the email array key is set
- This can be fixed and the test re-run

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

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

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

    if (empty($email)) {
        $valid = false;
    }

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

    return $valid;
}
```

PHPUnit

- When the phpunit is re-run it will run all the tests again
- You can see whether the bug has been fixed
- You can also see if making the change has introduced any new bugs



Automated testing

- Automated testing can be used to comprehensively test your application and locate bugs
- It takes time up front to write the code (usually there is more test code than the code being tested)
- but there are many advantages of unit tests

Advantages of unit testing

- Robustness: Each time you make a change, either minor or major all the tests can be very quickly re-run and check that no functionality is broken
  - Whenever you add new functionality you can add one or more tests for it
  - Instead of loading up the browser you can just run the tests which will test all the functionality at once
- Documentation: Someone else can look at the unit tests and see examples of how the function can be used

Disadvantages of unit testing

- Time: You have to spend time writing the tests
- Correctness: How do you ensure the tests are covering every possibility?

AssertEquals

- PHPUnit has many types of assertion, one useful assertion is assertEquals. It can be used like assertTrue and assertFalse but checks to see whether two values are the same

```
$this->assertEquals($value1,$value2);
```

Exercise 2

- Rewrite your test code from Exercise 1 to use PHPUnit

Exercise 2 - Solution

```php
<?php
require 'public_html/isPass.php';

class isPassTest extends \PHPUnit\Framework\TestCase {
  public function test20() {
    $result = isPass(20);
    $this->assertFalse($result);
  }

  public function test40() {
    $result = isPass(40);
    $this->assertTrue($result);
  }

  public function test60() {
    $result = isPass(60);
    $this->assertTrue($result);
  }
}
```

Exercise 3

- Save the supplied function in getGrade.php
- Save the test from the next side as testGetGrade.php
- This function gives a grade boundary for any given percentage e.g. C is 50-60%, A is 70+%
- Run PHPUnit to determine if there are any problems with this code
- Fix any bugs you find, keep re-running the test until no more bugs remain
- **You should not make any changes to the test code (except the file location if you store the function in a different folder)**

```
function getGrade($percentage) {
  if ($percentage > 40) {
    return 'D';
  }
  else if ($percentage > 50) {
    return 'C';
  }
  else if ($percentage > 60) {
    return 'B';
  }
  else if ($percentage > 70) {
    return 'A';
  }
}
```

```php
<?php
require 'public_html/getGrade.php';

class GetGradeTest extends \PHPUnit\Framework\TestCase {
  public function testA() {
    $result = getGrade(70);
    $this->assertEquals('A', $result);
  }

  public function testHighA() {
    $result = getGrade(99);
    $this->assertEquals('A', $result);
  }

  public function testB() {
    $result = getGrade(60);
    $this->assertEquals('B', $result);
  }

  public function testC() {
    $result = getGrade(50);
    $this->assertEquals('C', $result);
  }

  public function testD() {
    $result = getGrade(40);
    $this->assertEquals('D', $result);
  }
}
```

Exercise 4

- Extend the getGrade function to include an F grade for any percentage lower than 40.
- Add a test to check that the function still works
- Extend the getGrade function to include +/- grades.
  - D- < 44
  - D < 47
  - D+ < 50
  - C- < 54
  - C < 57
  - C+ < 60
  - Etc
- Include relevant tests to check for these grades

Exercise 5

- Write tests for the loadTemplate() function we wrote last week
  - You will need to create some templates that you can test
  - Use assertEquals to check that the function generates the correct result for given templates.
  - Hint: You should use some minimal templates just to check it works correctly

Exercise 6

- Complete the tests for the saveUser() function from this week