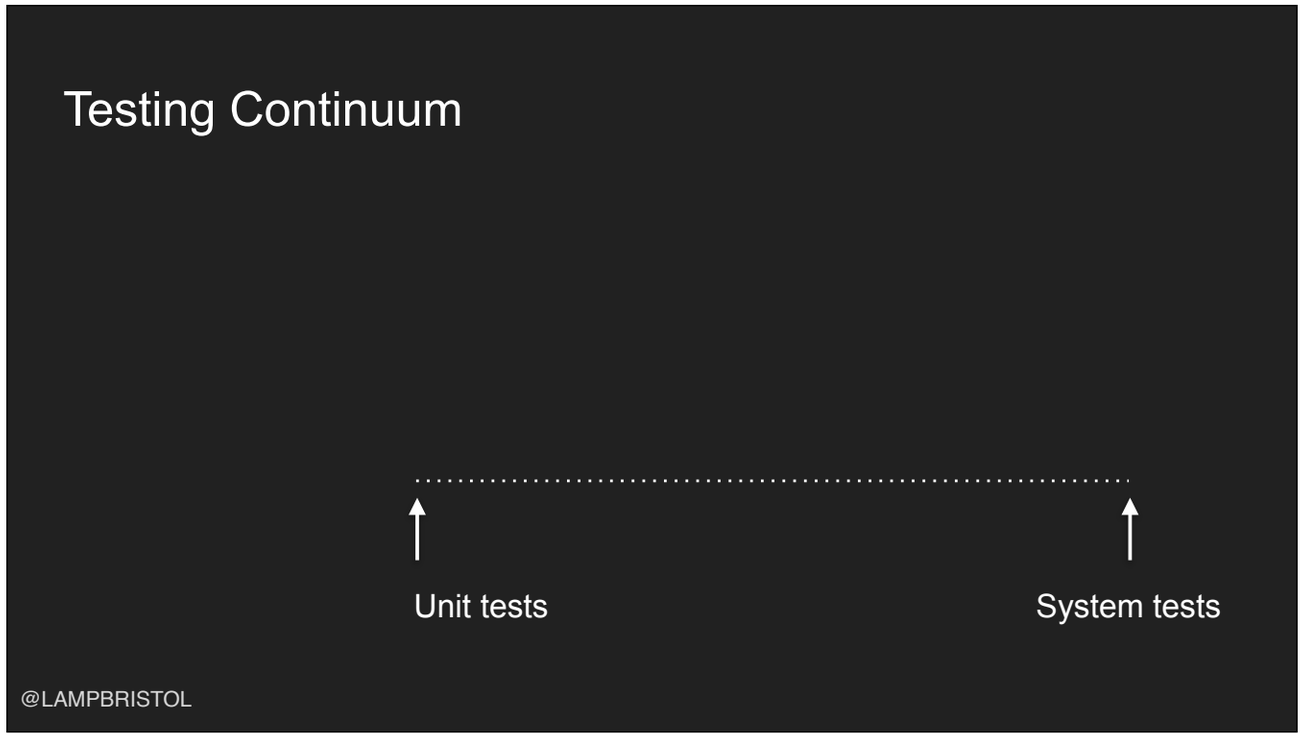

LAMP
Dave Liddament
Beyond Unit Testing
@PHPSW 10TH Feb 2016
@LAMPBRISTOL

Why test at all?

@LAMPBRISTOL

1) Check that code actually does the right thing.
2)Protect against regression when new features are added. How many times have you been working on a new feature and found that the you’ve broken existing functionality?
3)To allow us to refactor with confidence that you’ve not broken anything.

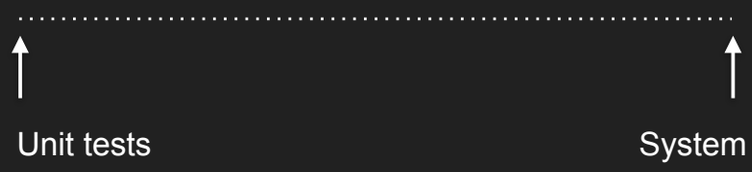


Today I'm only really talking about testing to establish if the software under test is functioning correctly. I'm not talking about load testing, performance testing, usability testing.

On the left we've got unit tests.
On the right we've got, what I'm calling system tests. We've got the software deployed on a staging/beta environment and a tester is manually testing it. As developers we often do this. We write a bit of code, fire up a web browser and check it all looks good.
Between these 2 extremes we've got something in the middle. People call the stuff in the middle all kinds of names (e.g. integration tests, functional tests - Google apparently break tests into small, medium and large). I'm not going to worry about the names apart from the 2 that appear on the slide.

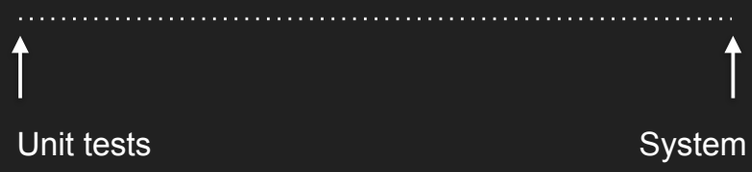
Testing Continuum

Speed of test execution Fast Slow



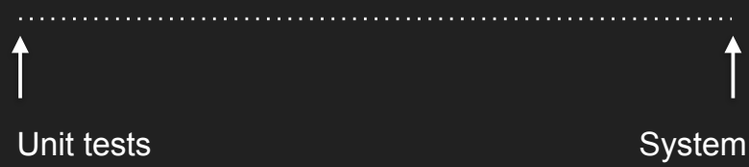
Testing Continuum

Speed of test execution	Fast	Slow
Speed of writing	Fast	Slow



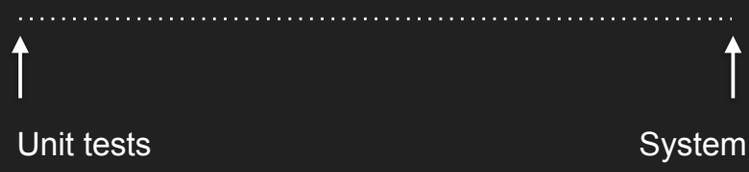
Testing Continuum

Speed of test execution	Fast	Slow
Speed of writing	Fast	Slow
Percentage automate	All	None



Testing Continuum

Speed of test execution	Fast	Slow
Speed of writing	Fast	Slow
Percentage automate	All	None
Robustness	Robust	Fragile



Why “Beyond Unit Tests”?

@LAMPBRISTOL

Unit tests have many advantages and we want to push as much testing as possible to the unit test level for the advantages we’ve outlined

But there is a whole area of testing that unit tests just can’t reach….

Password Validator

```
class PasswordValidator
{

        /**
         * Returns true if password is a valid one.
         *
         * A valid password meets the following criteria:
         * - 8 or more characters
         * - at least 1 digit
         * - at least 1 letter
         *
         * @param string $password
         * @return bool true if password meets criteria
         */
        public function isValid($password)
```

@LAMPBRISTOL

We want to write a unit test for this. This is simplest code you can write a unit test for. It is completely isolated and has no dependencies on any other classes.

Password Validator - New Requirement

```php
class PasswordValidator
{

        /**
         * Returns true if password is a valid one.
         *
         * A valid password meets the following criteria:
         * - 8 or more characters
         * - at least 1 digit
         * - at least 1 letter
         * - password is not one of the user's previous 5 passwords
         *
         * @param string $password
         * @param User $user
         * @return bool true if password meets criteria
         */
        public function isValid($password, User $user)
```


Now for a password to be valid we must make sure it's not a recently used one.

Password Validator - Getting Previous Passwords

```php
interface UserPasswordRepository
{

        /**
         * Returns the X most recent passwords for a user.
         *
         * @param int $numPasswords
         * @param User $user
         * @return string[] recent passwords
         */
        public function getRecentUserPasswords($numPasswords, User $user);

}
```

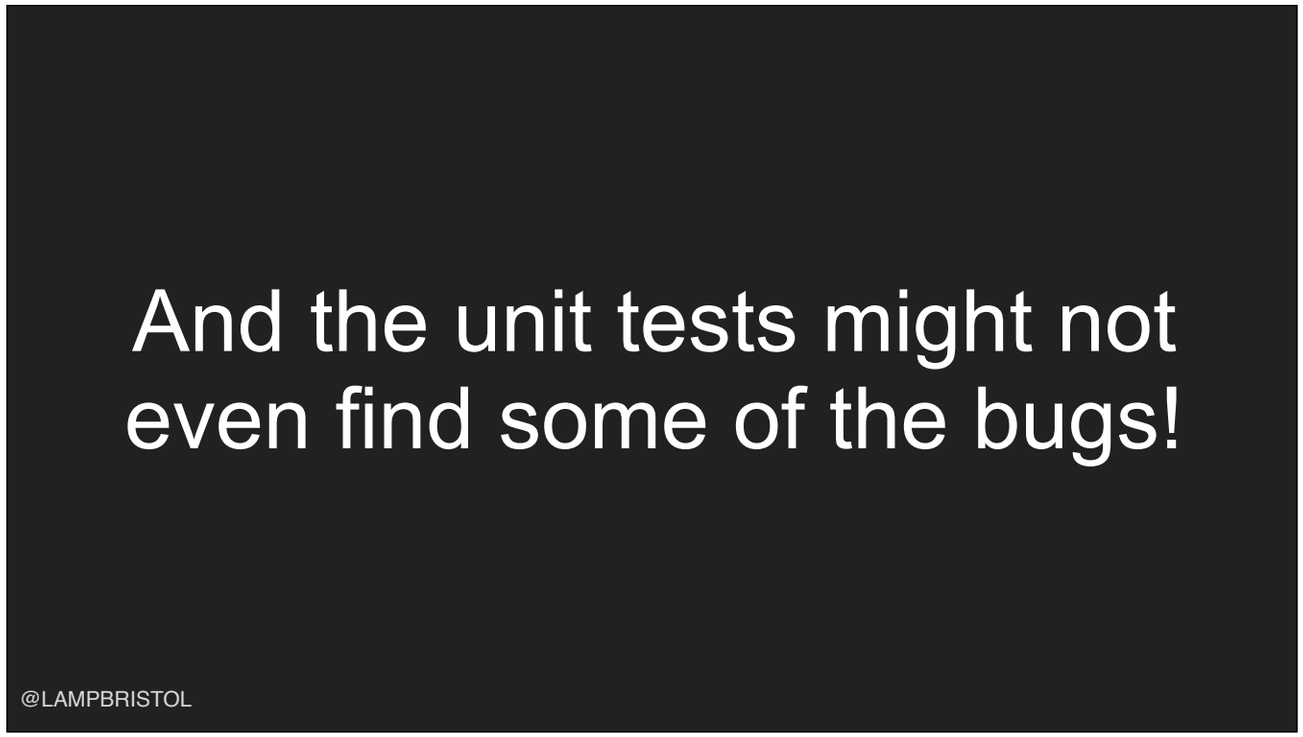

SRP (Single Responsibility Principle) in play here. PasswordValidator should only change if how we are validating a password changes.
How PasswordValidator gets recent password is not it’s concern, hence this is delegated out to UserPasswordRepository
This is an interface that could be used to get recent user passwords.

Our tests have got a bit more complicated…

@LAMPBRISTOL

Notice we've gone from a really simple unit test that was really quick to write to something that, is still a unit test, but a more complex one. The code under test is no longer run in isolation (it collaborates with AND is coupled to other objects). As a consequence the test case has to deal with mocking. It's not as quick to write. Test is slightly more fragile (due to coupling).

We're starting to see us moving a tiny step from unit test to the end to end test.



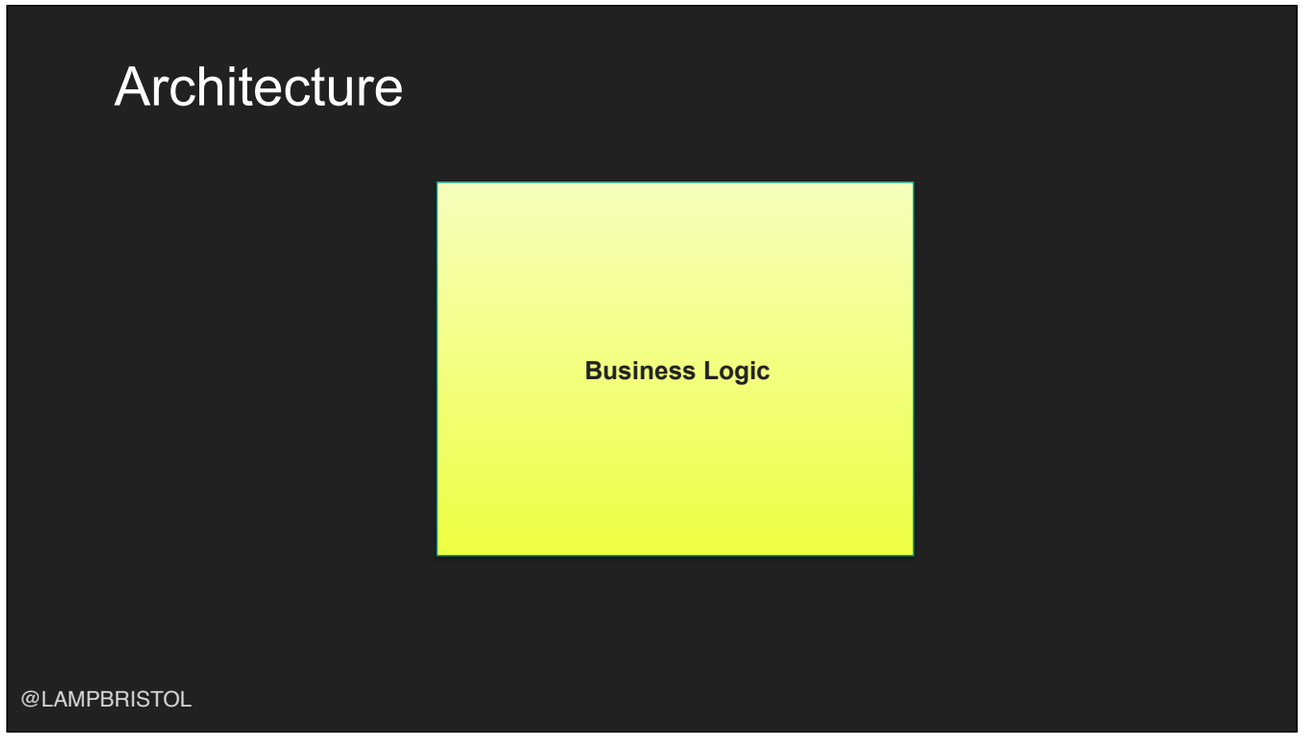
We could be so focused on testing that a password is not one of the user's previous 5 that we completely forget to implement the code to store a user's old password when they update to a new one. We might forget something crucial, like the fact we're storing hashed versions of the user's password in the database but the code we've written is comparing the plain text version of the password.

This is the limitation of unit tests. How do we know one bit of code will collaborate successfully with another bit? We can't with only unit tests.

How do we go beyond unit tests?

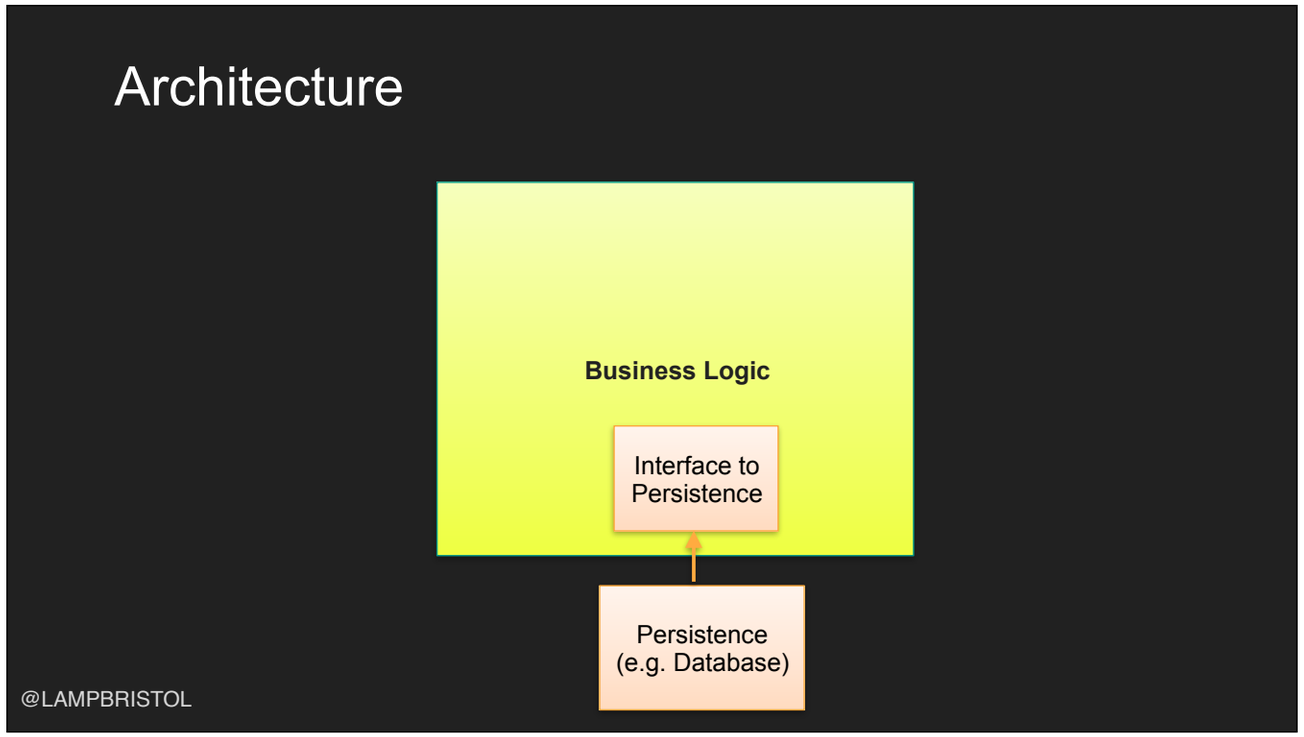
@LAMPBRISTOL

If you’re already doing good unit tests, you don’t need any more new skills or tools.
I use PHPUnit for all tests from unit all the way to system tests.
The choice of tool is NOT important. Pick one that works best for your team.
What is important is the architecture of your codebase. Follow basic OO principles: SRP, objects with high cohesion, low coupling between objects, design to interfaces



In the centre we have all our business logic. This is specific to whatever problem we're trying to solve. Composed of many classes. That are all interacting with each other.
The important bit here is that the business logic knows only of objects in this yellow box. It doesn't even know there is a world out there.

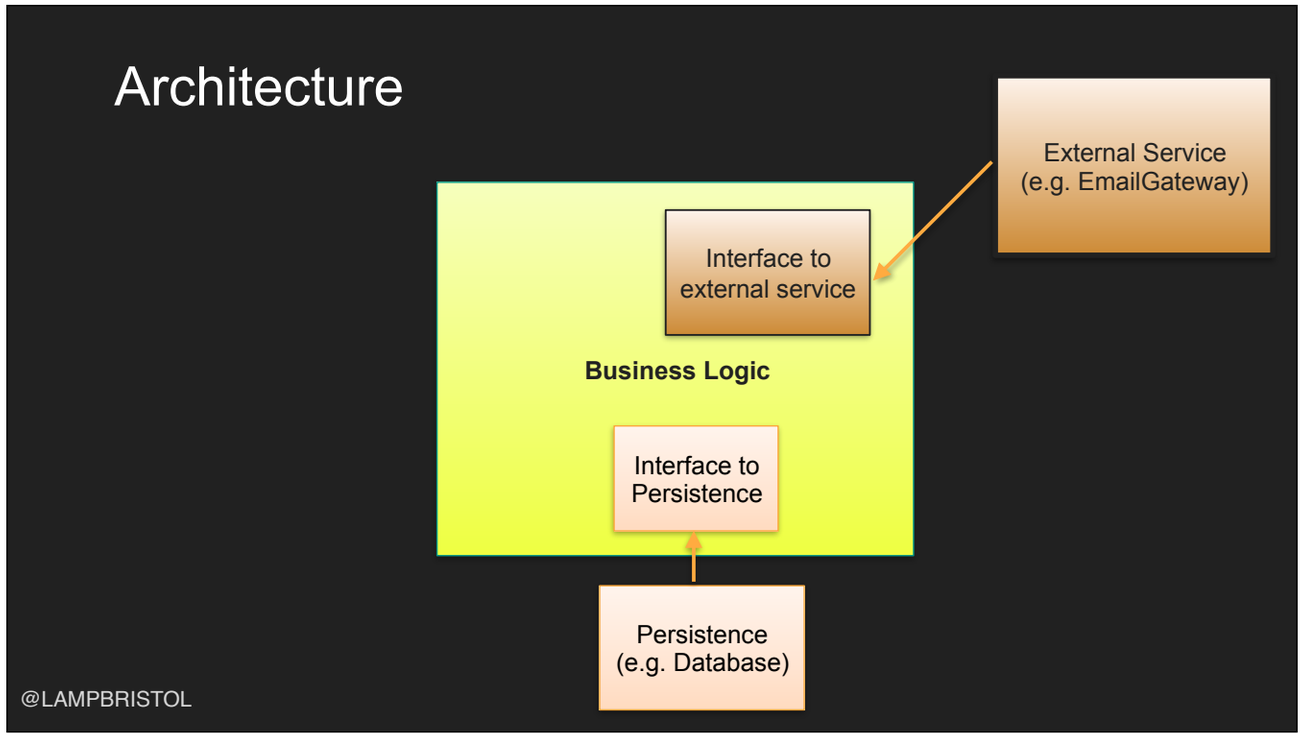
This is the area where most of our testing will be focuses. All operating at the unit or intermediate level of testings. All should be automated tests.



Will be some kind of persistence layer (database, filesystem). Again the BL only knows about the interface to the persistence layer. Not the actual implementation.

Persistence Layer Interface

```php
interface UserPasswordRepository
{

        /**
         * Returns the X most recent passwords for a user.
         *
         * @param int $numPasswords
         * @param User $user
         * @return string[] recent passwords
         */
        public function getRecentUserPasswords($numPasswords, User $user);

}
```



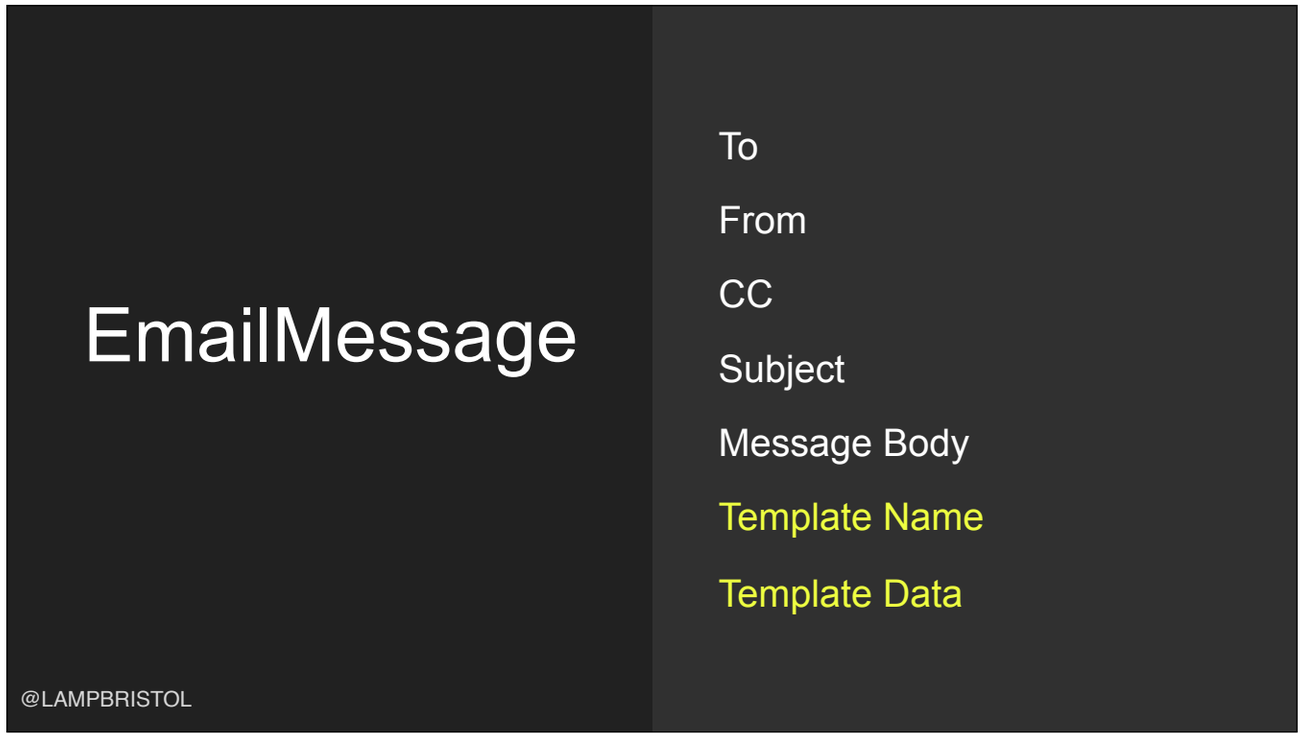
External services that the BL makes use of. E.g. Email Gateways, Payment Gateways, Twitter integration.

Again we the BL knows only of the interface.

Email Gateway Interface

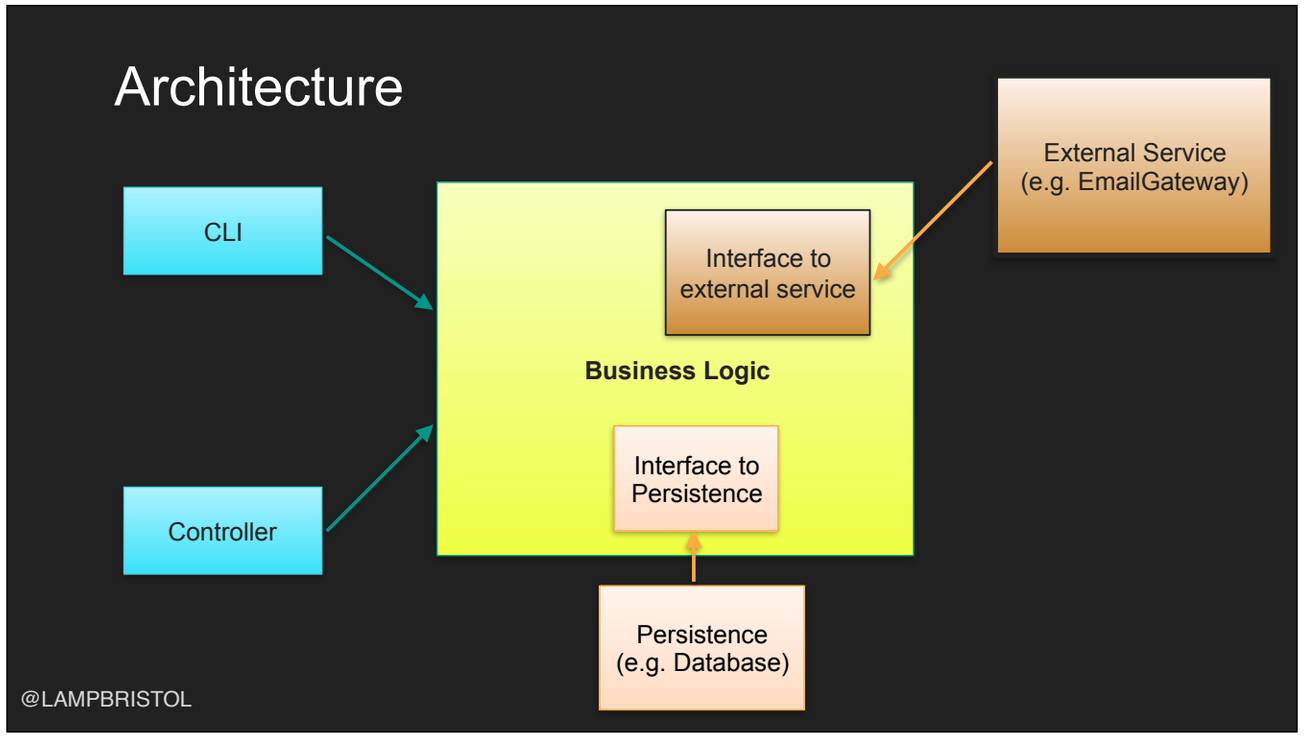
```php
interface EmailGatewayInterface
{
    /**
     * Gateway for sending and email
     *
     * @param EmailMessage $message to send
     */
     public function sendEmail(EmailMessage $message);
}
```


All external services are accessed by the Business Logic via an interface. This interface should be as simple as possible.



Fields that we store in the EmailMessage

The items in yellow are there to help us testing. Generally when constructing an email its a bit like rendering an HTML page. You have a template and data that the template references. Adding this to the EmailMessage makes testing easier as you'll see in a bit.



They'll be someway of getting information to and from the BL. HTTP requests. CLI. I'd use a framework for this.. Controllers should have no business logic at all. They should pull out data from the request and delegate to the business logic and then get the response from the BL and return via the view. Typically I use a service layer (which is a thin layer that provides an interface to the BL). Important the BL should no nothing of the framework.

Thin Controllers

```php
class UserController
{
        public function updatePassword()
        {
          $user = Auth::user();
          $newPassword = Input::get("newPassword");
          $success = $this->userService->updatePassword($user, $newPassword);

          if ($success) {
            // Handle success
          } else {
            // Handle failure
          }
        }
```

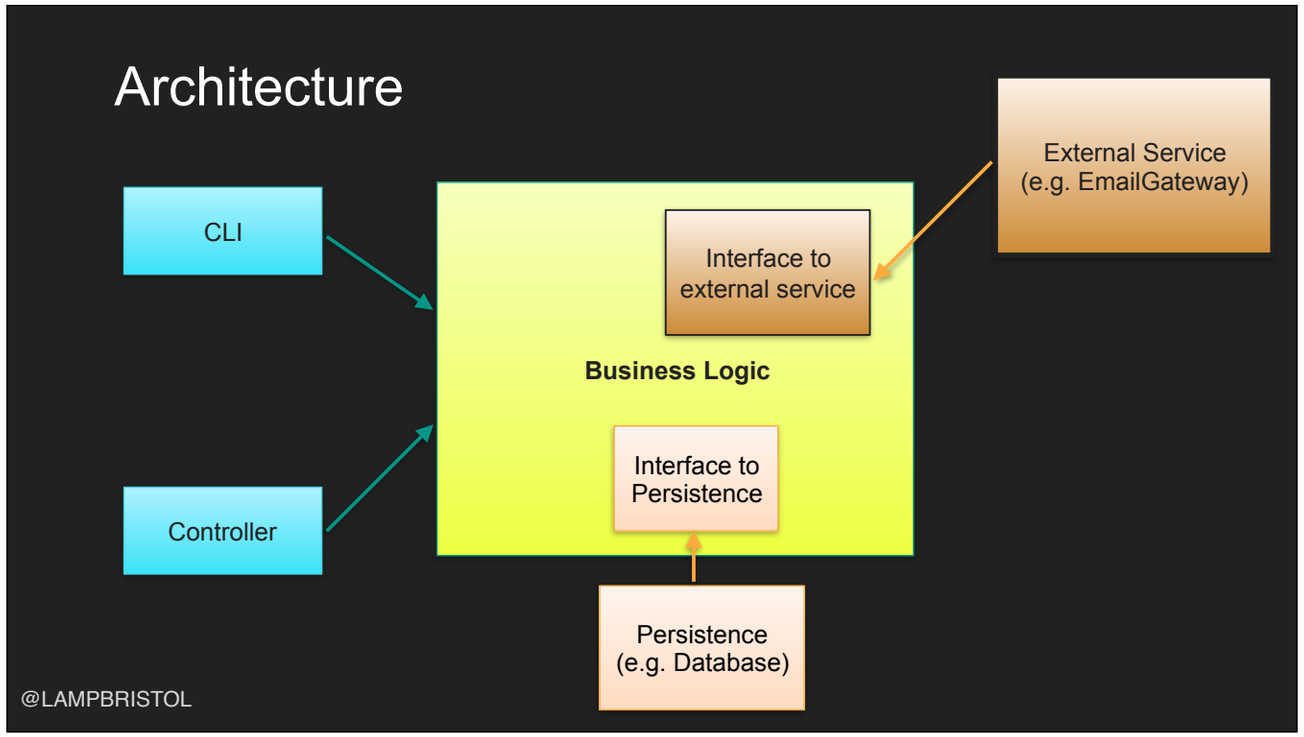

Thin controller (Laravel).

Responsible for extracting relevant information from HTTP request and then delegating to the business logic.
Service responds with success of true or false and then controller does what it needs to show user outcome of updating password

Thin Controllers

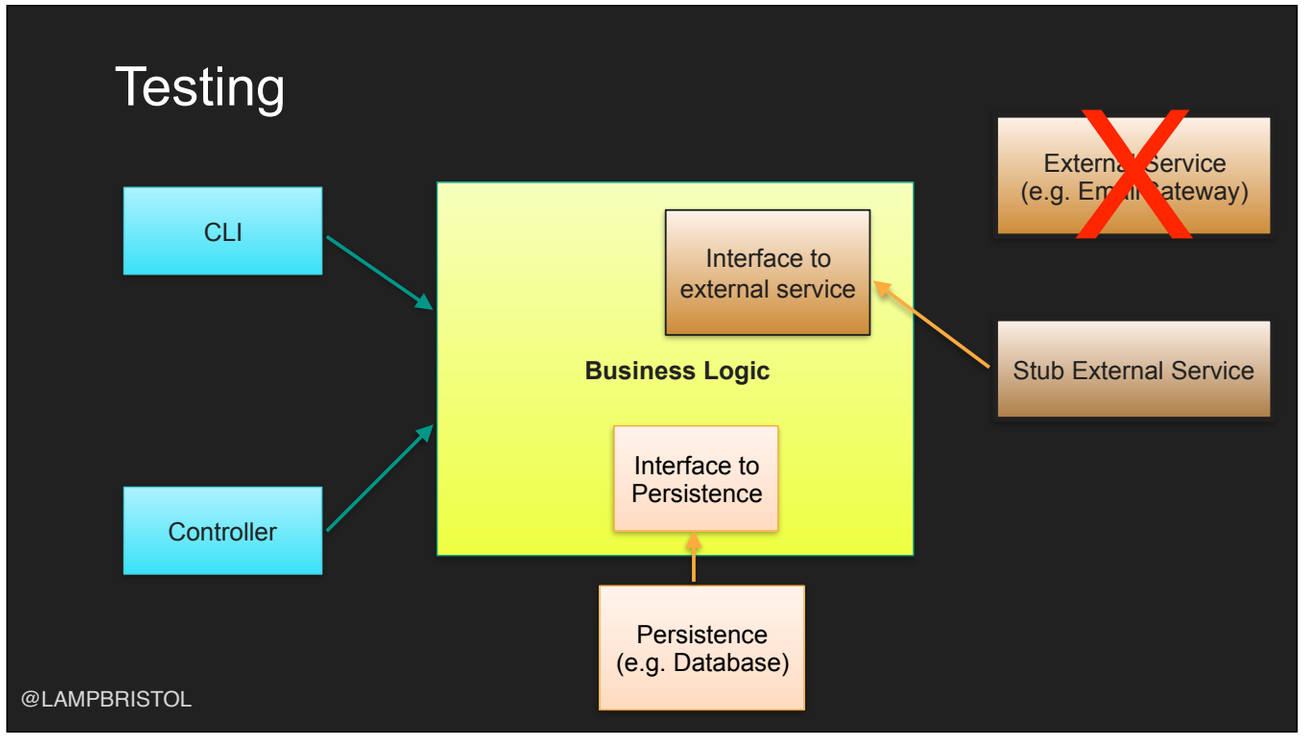
```
class UserController
{
        public function updatePassword()
        {
          $user = Auth::user();
          $newPassword = Input::get("newPassword");
          $success = $this->userService->updatePassword($user, $newPassword);

          if ($success) {
            // Handle success
          } else {
            // Handle failure
          }
        }
```


Emphasising delegating to the business logic via a service layer. The service layer is a thin layer that is the interface to the business logic.



Finally most of the code sits in some DI container. This is responsible for instantiating objects and resolving dependencies.

Other people draw this this as a hexagon and call this hexagonal architecture



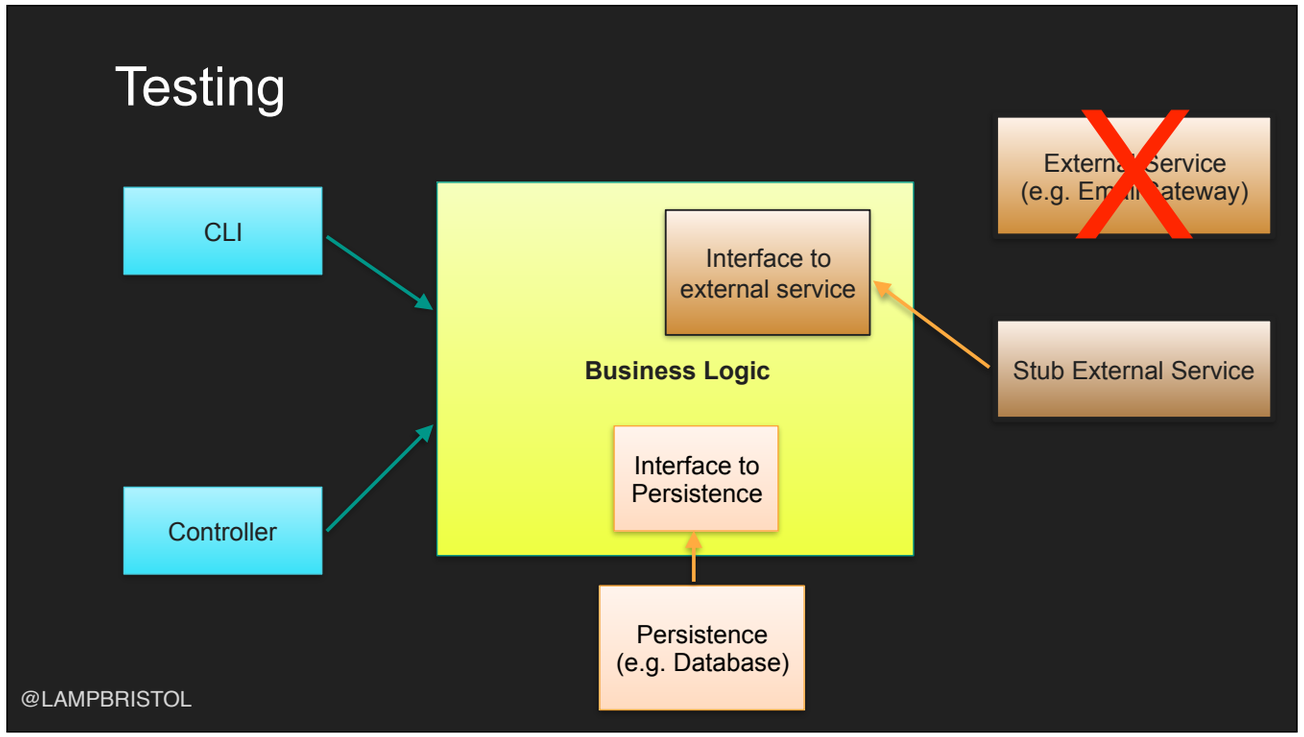
As with unit testing we need to provide fake implementations of external services, so we'll provide stubs for each of these.

Email Gateway Stub

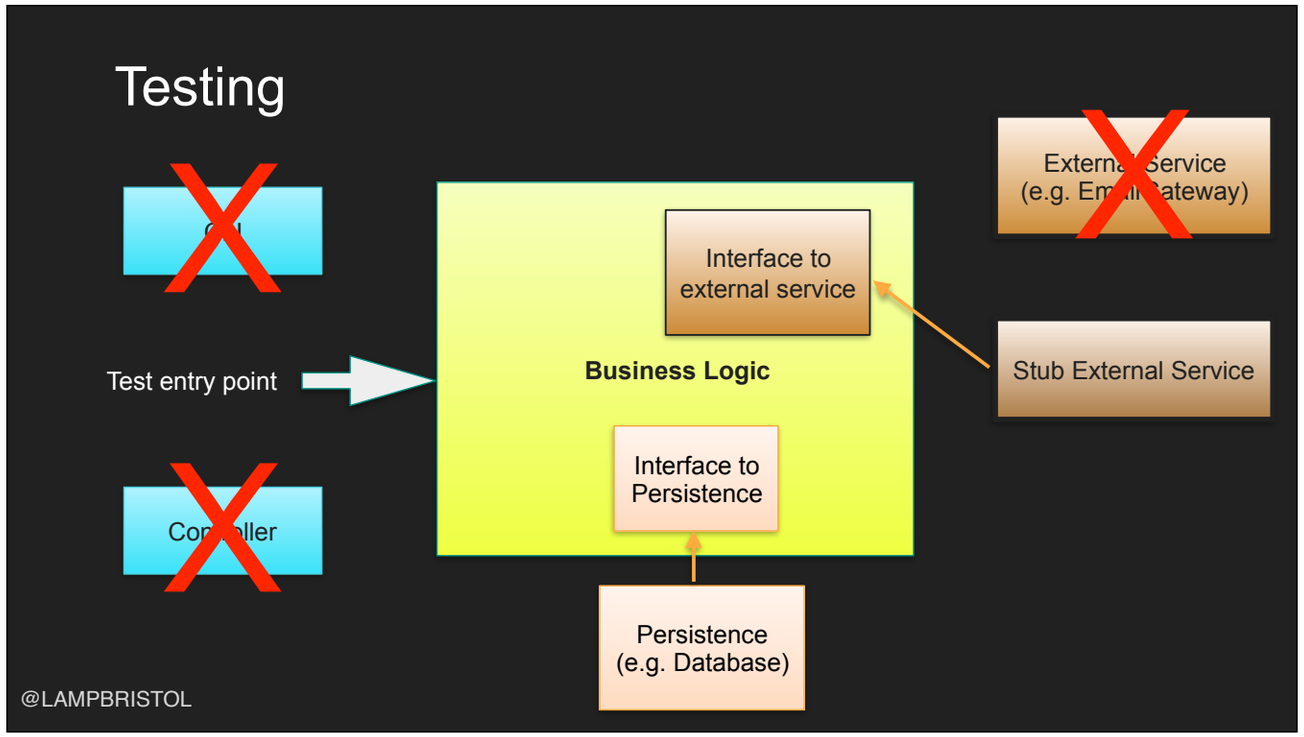
```php
class EmailGatewayStub implements EmailGatewayInterface
{
    public function sendEmail(EmailMessage $message)
    {
      /* implementation that stores all messages for searching */
    }

    /**
     * Find emails that would have been sent
     *
     * @param array $criteria e.g.:
     *      ['to' => 'dave@example.com', 'template' => 'RegisterUser']
     * @return EmailMessage[] messages that meet criteria
     */
     public function findEmails(array $criteria)

}
```


A stub implementation of the EmailGatewayInterface for testing.

sendEmail simply stores the EmailMessage object locally and these can be searched for later via the findEmails method.



Databases: I just use a real database. But before running either the test suite or even every test restore the database to a known state. This is one of the reasons that tests here are slower to run. There is the overhead of getting the database into a know state.



We know there is no business logic in the CLI or controllers so our main test entry point can be at the business logic. So our tests will be much faster and easier to write as we're dealing with code rather than parsing HTML and sending HTTP requests.

Testing Password Validator 1

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Get the UserService and register a new user
        $userService = $this->container->get("UserService");
        $userService->registerUser("dave@example.com", "1stPassword");

        // Get the EmailGatewayStub and find the registration email
        $emailGateway = $this->container->get("EmailGateway");
        $emails = $emailGateway->findEmails(
          ["to" => "dave@example.com", "template" => "RegisterUser"]);
        $this->assertEquals(1, count($emails));

        // Get confirmation token from the registration email
        $data = $emails[0]->getData();
        $confirmationToken = $data["confirmationToken"];

        // Complete registration
        $user = $userService->confirmUser($confirmationToken);
```


Let's write a test to check that the password validation works for previously used passwords.

Before we can write code to test updating a user's password to a previous password we'll first need to create a user.

You'll often find when running these kinds of integration tests that you have to run a number or steps to get the system into the right state for testing. This is another reason why these kinds of tests are slower to run. It's also are reason why these tests are more fragile (they depend on many steps and if any of those steps changes potentially the tests will break).

Testing Password Validator 1

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Get the UserService and register a new user
        $userService = $this->container->get("UserService");
        $userService->registerUser("dave@example.com", "1stPassword");

        // Get the EmailGatewayStub and find the registration email
        $emailGateway = $this->container->get("EmailGateway");
        $emails = $emailGateway->findEmails(
          ["to" => "dave@example.com", "template" => "RegisterUser"]);
        $this->assertEquals(1, count($emails));

        // Get confirmation token from the registration email
        $data = $emails[0]->getData();
        $confirmationToken = $data["confirmationToken"];

        // Complete registration
        $user = $userService->confirmUser($confirmationToken);
```


Let's imagine that as part of completing registering a new user is clicking a link in an email.

Here we can see how easy it is to assert that a 'Register User' email was sent.
We could have done this by looking at the subject heading of an email but that would lead to fragile tests. An email subject heading is far more likely to change than the template used (e.g. if we introduced internationalisation).

Testing Password Validator 1

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Get the UserService and register a new user
        $userService = $this->container->get("UserService");
        $userService->registerUser("dave@example.com", "1stPassword");

        // Get the EmailGatewayStub and find the registration email
        $emailGateway = $this->container->get("EmailGateway");
        $emails = $emailGateway->findEmails(
          ["to" => "dave@example.com", "template" => "RegisterUser"]);
        $this->assertEquals(1, count($emails));

        // Get confirmation token from the registration email
        $data = $emails[0]->getData();
        $confirmationToken = $data["confirmationToken"];

        // Complete registration
        $user = $userService->confirmUser($confirmationToken);
```


Again we've made our life easy by suppling the data used to create the email message.

Without this we'd have had to parse an email to grab the confirmation token.

Testing Password Validator 1

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Get the UserService and register a new user
        $userService = $this->container->get("UserService");
        $userService->registerUser("dave@example.com", "1stPassword");

        // Get the EmailGatewayStub and find the registration email
        $emailGateway = $this->container->get("EmailGateway");
        $emails = $emailGateway->findEmails(
          ["to" => "dave@example.com", "template" => "RegisterUser"]);
        $this->assertEquals(1, count($emails));

        // Get confirmation token from the registration email
        $data = $emails[0]->getData();
        $confirmationToken = $data["confirmationToken"];

        // Complete registration
        $user = $userService->confirmUser($confirmationToken);
```

Testing Password Validator 2

```
// Update password to a valid one
$success = $userService->updatePassword($user, "2ndPassword");
$this->assertTrue($success);

// Update password to first password
$success = $userService->updatePassword($user, "1stPassword");
$this->assertFalse($success);
```


Now everything is setup we are in a position where we can go through and test the scenarios relating to updating passwords that we couldn't do at unit test level.

1st we update password to a new one (this should work)

Testing Password Validator 2

```
// Update password to a valid one
$success = $userService->updatePassword($user, "2ndPassword");
$this->assertTrue($success);

// Update password to first password
$success = $userService->updatePassword($user, "1stPassword");
$this->assertFalse($success);
```


Then we try an update password to the first one. This should fail.

Recap

Between unit and system test

Validated correct behaviour

Automated test (big win)

Faster than manual testing

Test that is much closer to user stories

Well architected code

So we've just made a test that sits somewhere between unit and system level.
Now when we release our code we'll be pretty sure that our PasswordValidator really works.
The test is automated, this means it can get run by a CI server EVERY commit. This protects us against regression. This is a massive win
The test will run much faster than manual test.
Also the test is much closer to a user story. These are really useful and important tests as they validate that the code does what the user stories require.
If it's easy to write these tests they your code is probably well architected.

If you think about tasks like registering a user, we'll have many tasks like that. We want to wrap these actions into code we can reuse. 3 reasons:
- DRY
- Encapsulate the task. If the user registration process changes we've only got to update 1 class, rather than every test. This makes our tests more robust.
- We'll build up chunks of code that we can reuse to help us write tests quicker.

Tidy up: Use Command Pattern

```php
interface Command
{
    public function execute(Container $container);
}
```


All our tasks could be commands.

Tidy up: RegisterUserCommand

```
class RegisterUserCommand implements Command
{
   public function __construct($email, $password)
    {
        $this->email = $email;
        $this->password = $password;
    }

    public function execute(Container $container)
    {
        // Code to fully register a new user (see previous slide)
        $this->user = …
    }

    public function getUser()
    {
      return $this->user;
    }
```


This is very similar to the PageObject pattern used in front end testing.

Tidy up: Cleaner test

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Register a new user
        $registerUserCommand = new RegisterUserCommand(
            “dave@example.com”, “1stPassword");
        $registerUserCommand->execute($this->container);
         $user = $registerUserCommand->getUser();

        // Tests as before

        // Update password to a valid one
        $success = $userService->updatePassword($user, “2ndPassword");
        $this->assertTrue($success);

        // Update password to the first password
        $success = $userService->updatePassword($user, “1stPassword");
        $this->assertFalse($success);
```


Now all the setup for our test is hidden away in the RegisterUserCommand so we don’t have to worry about it.

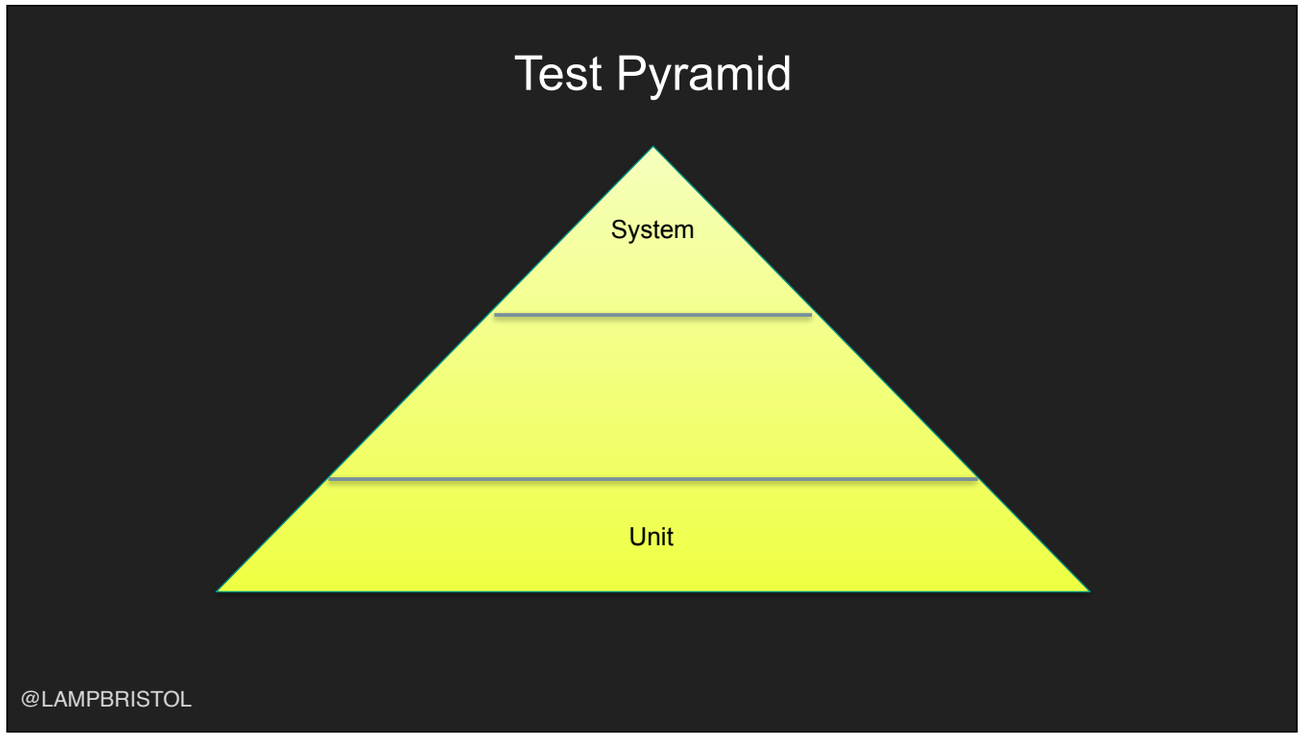
Tidy up: Cleaner test

```
class PasswordValidatorTest extends AbstractTestCase
{
  public function testUpdatePassword()
  {
        // Register a new user
        $registerUserCommand = new RegisterUserCommand(
            "dave@example.com", "1stPassword");
        $registerUserCommand->execute($this->container);
         $user = $registerUserCommand->getUser();

        // Tests as before

        // Update password to a valid one
        $success = $userService->updatePassword($user, "2ndPassword");
        $this->assertTrue($success);

        // Update password to the first password
        $success = $userService->updatePassword($user, "1stPassword");
        $this->assertFalse($success);
```


The actual test code is as before.

So over time we'll build up a series of commands that will allow us to write tests reasonably quickly.

Also you'll find that you'll have 'super' commands that are composed of a number of normal commands.



We want to do as much of our testing at unit test level as this is easiest and quickest place to do it.

Consider our PasswordValidator test:
- At unit level we do several tests (password valid, password too short, password missing digit, etc)
- At the intermediate level we have 3 tests (password valid, password invalid, duplicate password)
- At system level we have 2 tests (password valid, password invalid)

Summary

Need more than unit tests

Automate tests if possible

Think about your architecture

Push tests down the pyramid

https://joind.in/16989

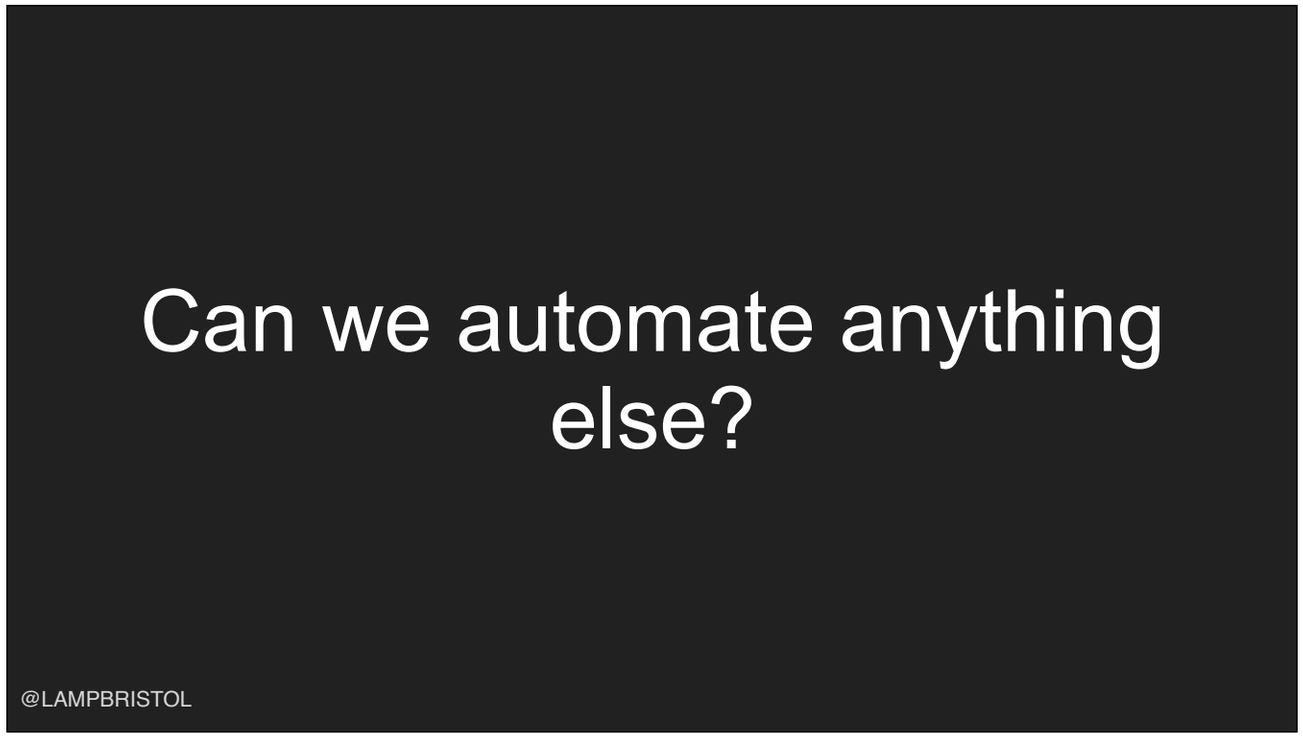
@LAMPBRISTOL

Questions and feedback (please be nice!)

Bonus slides

@LAMPBRISTOL

If we have time…



Sort of. Let’s consider what to do with the real implementation of the EmailGateway, the one that actually sends the email.

We can write a script that calls the real EmailGateway code and sends us an email. So before every release someone could run through each of the test scripts and check that the correct action has actually happened. E.g. in case of EmailGateway an email has been sent.
Payment gateway a payment has been made. And so on.

Automating as much as we can:

```
php bin/console test:emailgateway --to dave@lampbristol.com

Sending email:
To      [dave@lampbristol.com]
From    [test@lampbristol.com]
CC      [dave+1@lampbristol.com]
Subject [Test email 2016-02-08 19:37]
Body    [Hi,
         This is a test email.
         Sent at 2016-02-08 19:37.
         From your tester]
```


Most frameworks have a CLI interface (this one is Symfony 3).
Run the command (in orange).
It tells you what it has done and you can check you receive an email as it specifies.

So we've automated as much as we can and minimised what the human tester has to do.