Codeception Introduction

Sílvio Priem Mendes, PhD

Cofinanciado por:

POCH
PORTUGAL 2020
UNIÃO EUROPEIA
Fundo Social Europeu

- The idea behind testing is not new. Having your application covered with tests gives you more trust in the stability of your application. **That’s all**.
- In most cases tests don’t guarantee that the application works 100% as it is supposed to. You can’t predict all possible scenarios and exceptional situations for complex apps, but with tests you can cover the most important parts of your app and at least be sure they work as predicted.

Why Testing?

- There are plenty of ways to test your application. The most popular paradigm is **Unit Testing**.
- For web applications, testing just the controller and/or the model doesn't prove that your application is working. To test the behavior of your application as a whole, you should write **functional** or **acceptance** tests.
- Codeception supports all three testing types. Out of the box you have tools for writing unit, functional, and acceptance tests in a unified framework.

Why Testing?

- Codeception tries to simplify and aggregate the process of writing tests, plugging different testing suites. Codeception has support for:
    - **Symfony**
    - **Joomla**
    - **Laravel**
    - **Wordpress**
    - **Yii2**
    - **Zend 2**

Use Cases

- Follow the composer installation steps at:
  - http://codeception.com/install
- And bootstrap codeception

Installation

- Intuitively, one can view a unit as the smallest testable part of an application.
- In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure.
- In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.
- Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process.

Unit Testing

- Ideally, each **test case** is independent from the others.
- Substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in isolation.
- Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended.

Unit Testing

- Testing pieces of code before coupling them together is highly important as well. This way, you can be sure that some deeply hidden feature still works, even if it was not covered by functional or acceptance tests. This also shows care in producing stable and testable code.
- Codeception is created on top of **PHPUnit**. If you have experience writing unit tests with PHPUnit you can continue doing so.
- Codeception has no problem executing standard PHPUnit tests, but, additionally, Codeception provides some well-built tools to make your unit tests simpler and cleaner.

Unit Testing

- Requirements and code can change rapidly, and unit tests should be updated every time to fit the requirements. The better you understand the testing scenario, the faster you can update it for new behavior.

Unit Testing

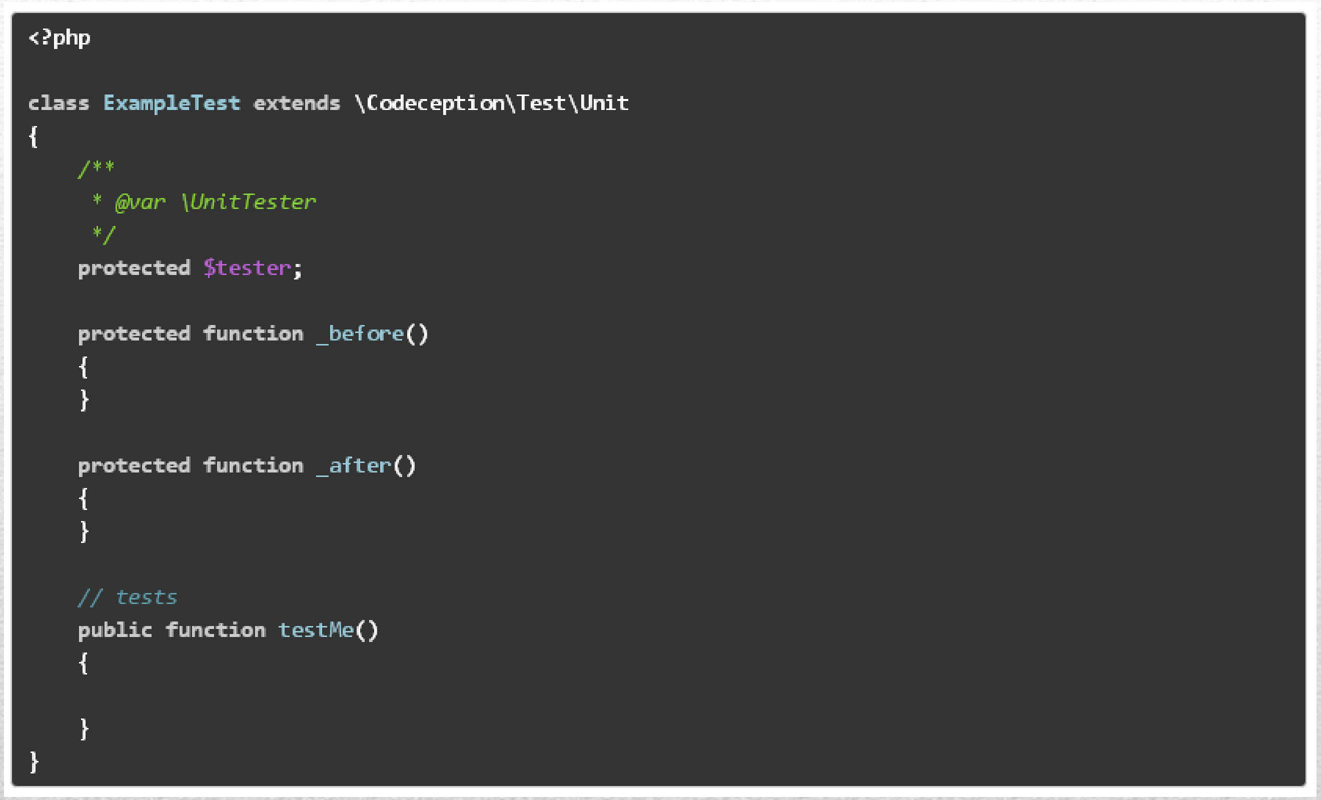
- Create a test using `generate:test` command with a suite and test names as parameters:
  - `php codecept generate:test unit ExampleTest`
- Creates a new ExampleTest file located in the tests/unit directory.

Codeception Unit Testing

- As always, you can run the newly created test with this command:
  - `php codecept run unit ExampleTest`
- Or simply run the whole set of unit tests with:
  - `php codecept run unit`

Codeception Unit Testing

- A test created by the `generate:test` command will look like this:



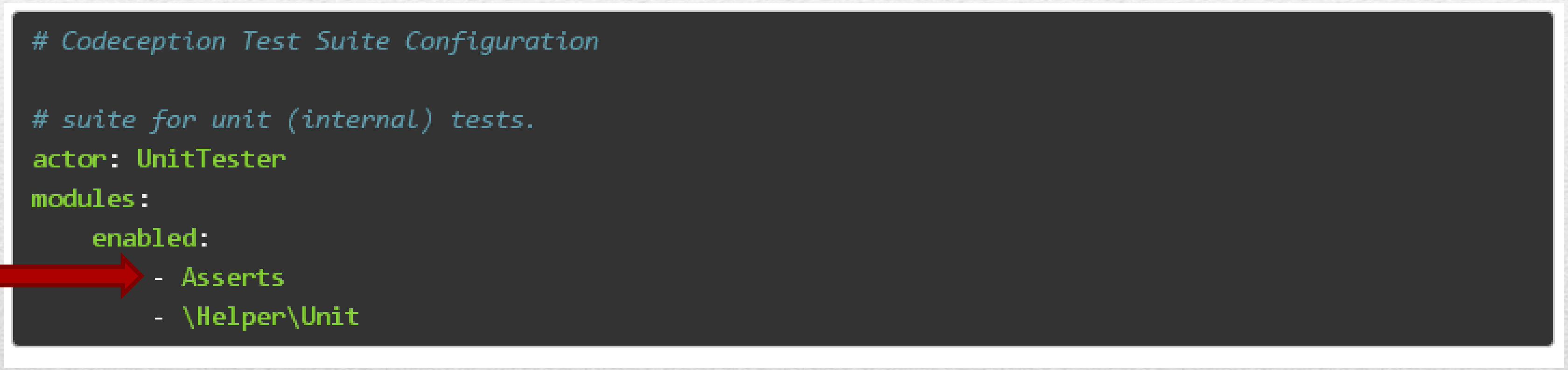
```php
<?php

class ExampleTest extends \Codeception\Test\Unit
{
    /**
     * @var \UnitTester
     */
    protected $tester;

    protected function _before()
    {
    }

    protected function _after()
    {
    }

    // tests
    public function testMe()
    {

    }
}
```

Codeception Unit Testing

- This class has predefined `_before` and `_after` methods. You can use them to create a tested object before each test, and destroy it afterwards.
- As you see, unlike in PHPUnit,
  the `setUp` and `tearDown` methods are replaced with their aliases: `_before`, `_after`.
- The actual `setUp` and `tearDown` are implemented by the parent class `\Codeception\TestCase\Test`

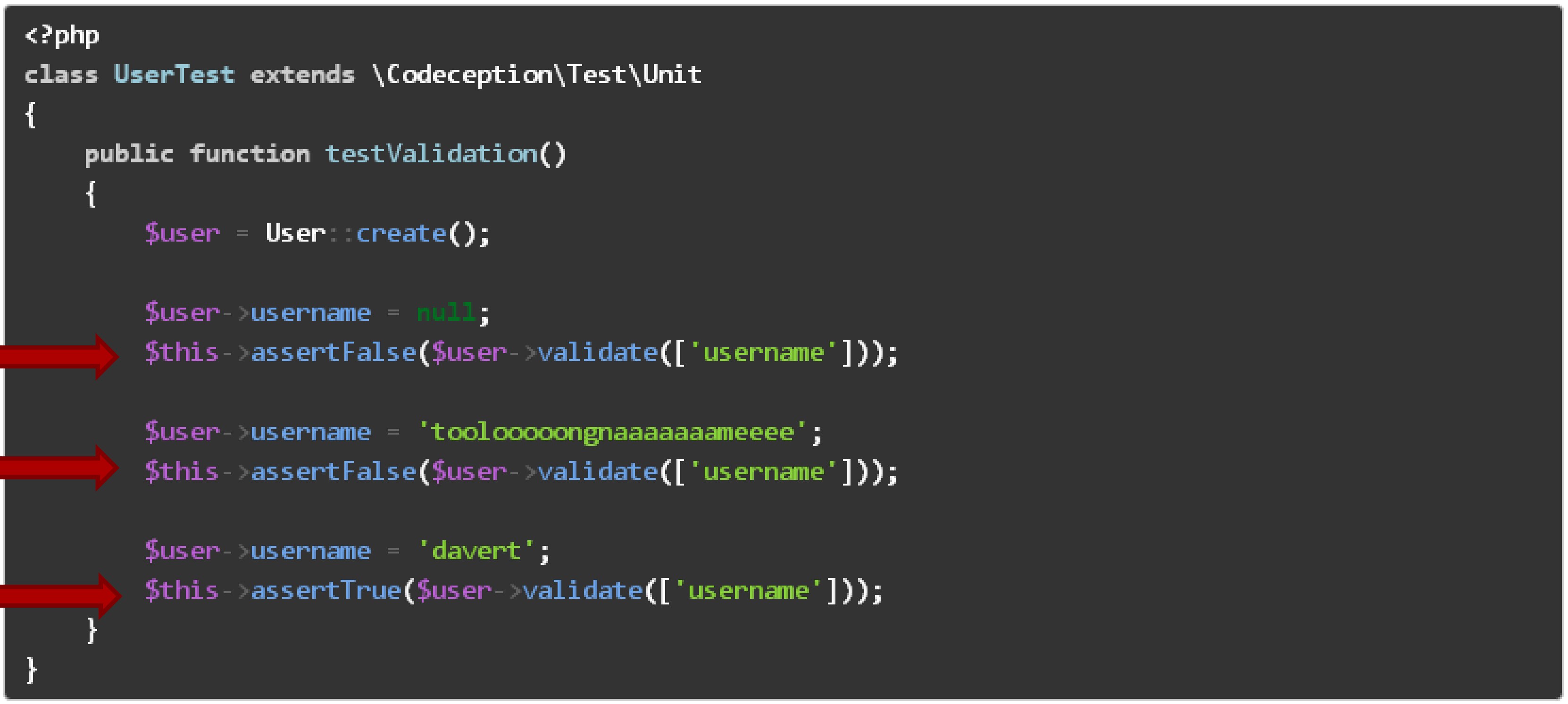
Codeception Unit Testing

- you can choose the proper modules for the `UnitTesterclass` in the `unit.suite.yml` configuration file:



```
# Codeception Test Suite Configuration

# suite for unit (internal) tests.
actor: UnitTester
modules:
    enabled:
        - Asserts
        - \Helper\Unit
```

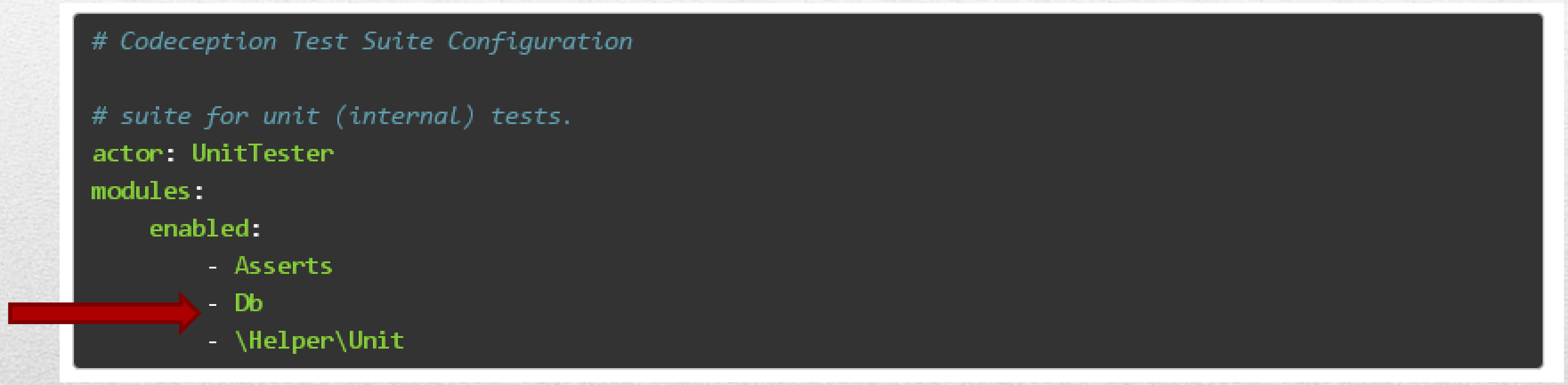
Codeception Modules



```php
<?php
class UserTest extends \Codeception\Test\Unit
{
    public function testValidation()
    {
        $user = User::create();

        $user->username = null;
        $this->assertFalse($user->validate(['username']));

        $user->username = 'toolooooongnaaaaaaameeee';
        $this->assertFalse($user->validate(['username']));

        $user->username = 'davert';
        $this->assertTrue($user->validate(['username']));
    }
}
```

Classical Unit Testing

- If you write integration tests, it may be useful to include the **Db** module for database testing.

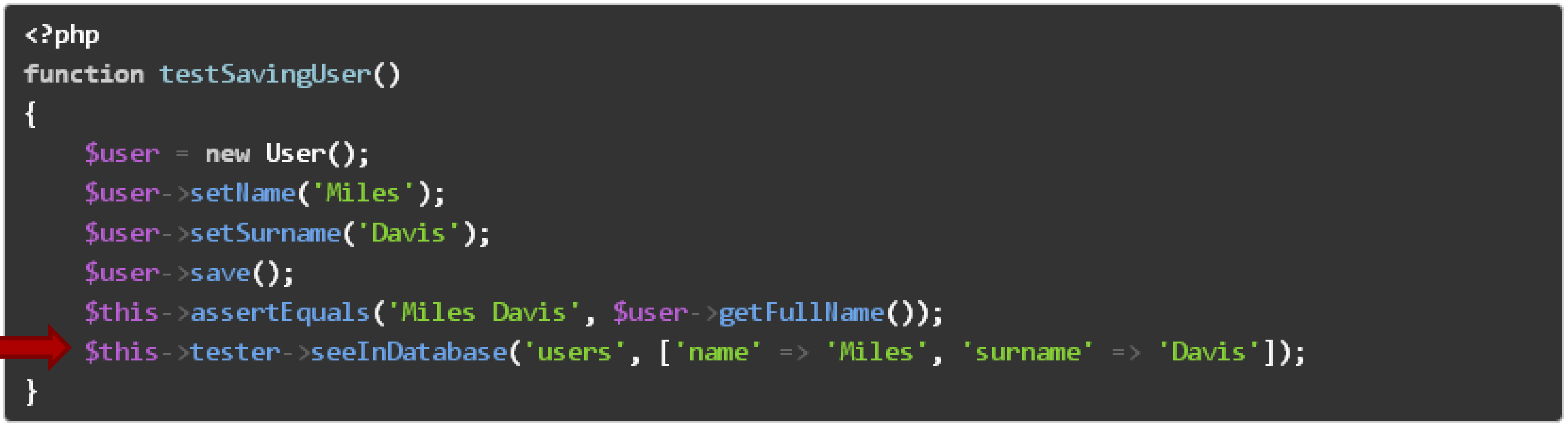


```
# Codeception Test Suite Configuration

# suite for unit (internal) tests.
actor: UnitTester
modules:
    enabled:
        - Asserts
        - Db
        - \Helper\Unit
```

Codeception Modules

- To enable the database functionality in unit tests, make sure the `Db` module is included in the `unit.suite.yml` configuration file.
- The database will be cleaned and populated after each test.
- If that’s not your required behavior, change the settings of the `Db` module for the current suite.

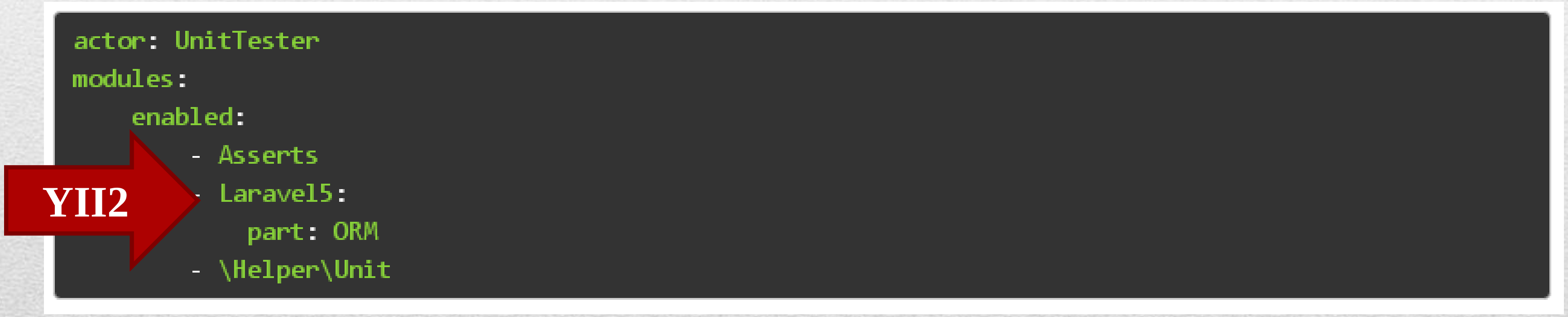
Testing Database



```php
<?php
function testSavingUser()
{
    $user = new User();
    $user->setName('Miles');
    $user->setSurname('Davis');
    $user->save();
    $this->assertEquals('Miles Davis', $user->getFullName());
    $this->tester->seeInDatabase('users', ['name' => 'Miles', 'surname' => 'Davis']);
}
```

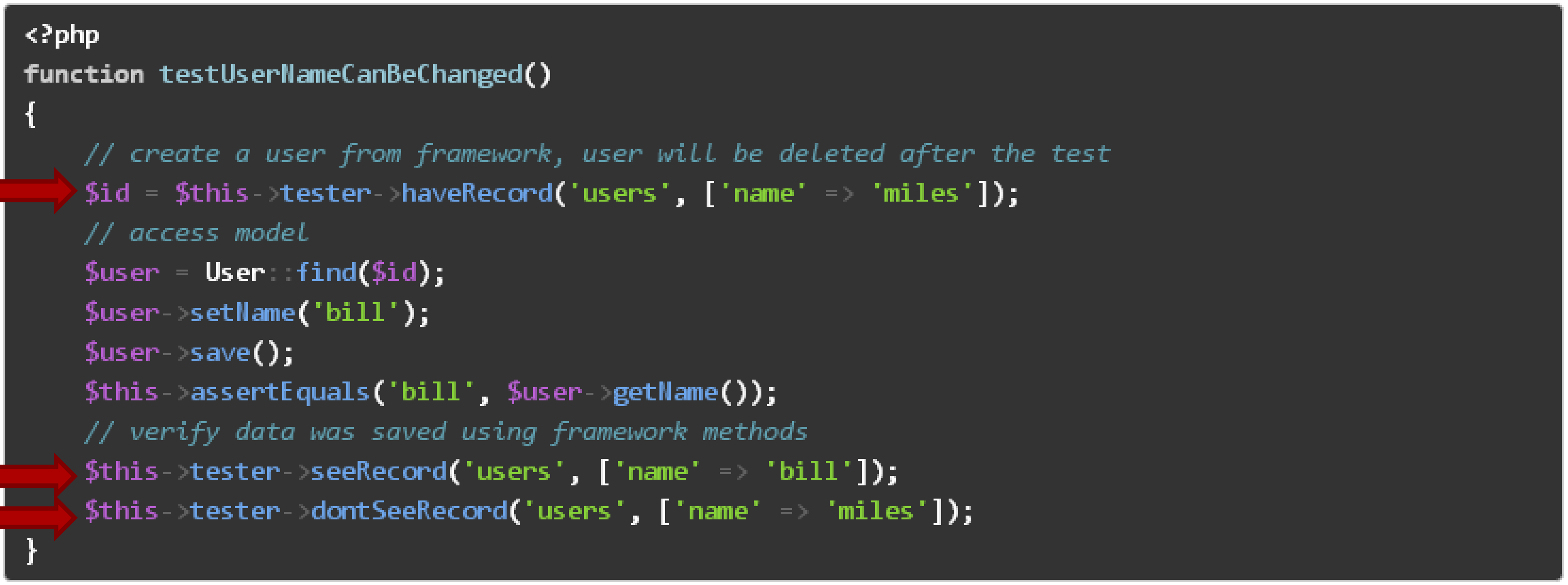
Database (Unit) Testing

- You should probably not access your database directly if your project already uses ORM for database interactions. Why not use ORM directly inside your tests?



```yaml
actor: UnitTester
modules:
    enabled:
        - Asserts
        - Laravel5:
            part: ORM
        - \Helper\Unit
```

Framework Interaction



```php
<?php
function testUserNameCanBeChanged()
{
    // create a user from framework, user will be deleted after the test
    $id = $this->tester->haveRecord('users', ['name' => 'miles']);
    // access model
    $user = User::find($id);
    $user->setName('bill');
    $user->save();
    $this->assertEquals('bill', $user->getName());
    // verify data was saved using framework methods
    $this->tester->seeRecord('users', ['name' => 'bill']);
    $this->tester->dontSeeRecord('users', ['name' => 'miles']);
}
```

- In Yii2, the methods `haveRecord`, `seeRecord`, `dontSeeRecord` work in the same way.

Framework Interaction

- When developing DB integration (unit) tests consider all **CRUD** methods in the following order:
  - `Create data test`
  - `Read data test`
  - `Update data test`
  - `Delete data test`

Database Integration