



# PEST

Bring back the joy of testing in PHP

Tech and Beer #2, 31st May 2023

Sponsored by

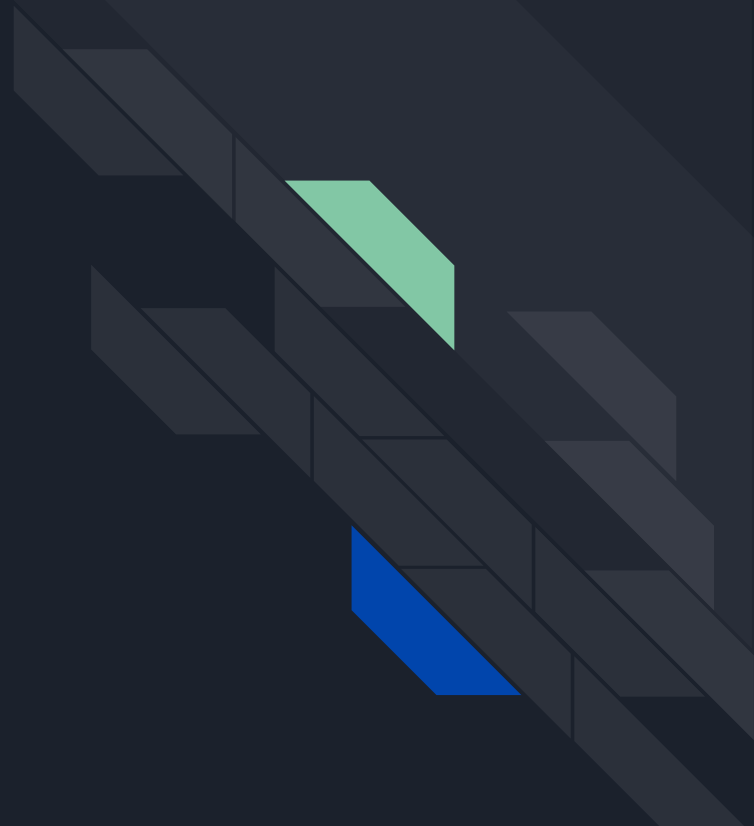
PTER!

```
{  
  "name": "Claudio La Barbera",  
  "role": "Full Stack Developer",  
  "company": "Angelini Consumer",  
  "linkedin": "https://www.linkedin.com/in/claudio-la-barbera",  
  "github": "https://github.com/thebatclaudio"  
}
```

# What is PEST?

*Pest is a testing framework with a focus on **simplicity**, meticulously designed to bring back the joy of testing in PHP.*

[pestphp.com](https://pestphp.com)



# PEST is built on top of PHPUnit!

Your current PHPUnit test suite will work flawlessly with Pest. No need to change a thing.





# Installation

*First step: require Pest as a "dev" dependency on your project:*

```
composer require pestphp/pest --dev --with-all-dependencies
```

*Second step: initialize Pest on your project:*

```
./vendor/bin/pest --init
```

*Finally: you can run your tests:*

```
./vendor/bin/pest
```

*If you are using Laravel you can also run:*

```
php artisan test
```



# Installation

*Here is an example of the output in a new Laravel project:*

```
→ ./vendor/bin/pest
```

```
PASS Tests\Unit\ExampleTest
```

```
✓ that true is true
```

```
PASS Tests\Feature\ExampleTest
```

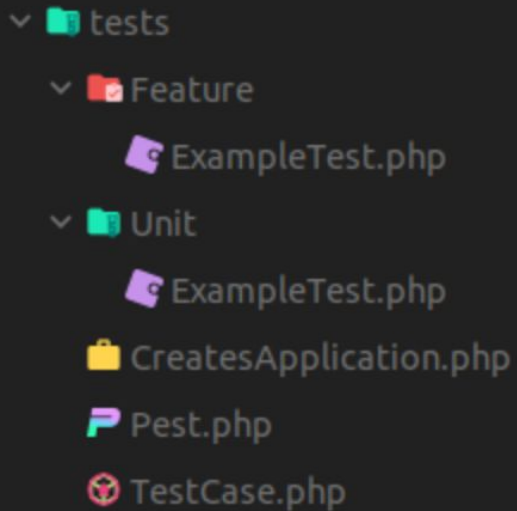
```
✓ it returns a successful response
```

```
Tests:    2 passed (2 assertions)
```

```
Duration: 0.09s
```

# Installation

*Here is an example of the folder structure in a new Laravel project:*



# From PHPUnit to Pest: PHPUnit

php ExampleTest.php

```
<?php

namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     */
    public function test_that_true_is_true(): void
    {
        $this->assertTrue(true);
    }
}
```





# From PHPUnit to Pest: Pest

php ExampleTest.php

```
<?php
```

```
test("true is true", function () {  
    $this->assertTrue(true);  
});
```

Writing tests





# test() function

ExampleTest.php

```
<?php
```

```
test("true is true", function () {  
    $this->assertTrue(true);  
});
```



# test() function

ExampleTest.php

```
<?php
```

```
test("false is not true", function () {  
    $this->assertNotTrue(false);  
});
```



# it() function

ExampleTest.php

```
<?php
```

```
it("is not true", function () {  
    $this->assertNotTrue(false);  
});
```



# skip() function

ExampleTest.php

```
<?php
```

```
it("has homepage", function () {  
    // homepage does not exist yet  
})→skip();
```



# skip() function

**PASS** Tests\Unit\ExampleTest

✓ true is true

**WARN** Tests\Feature\ExampleTest

- it has homepage

Tests: **1 skipped**, **1 passed** (1 assertions)

Duration: 0.09s



# skip() function

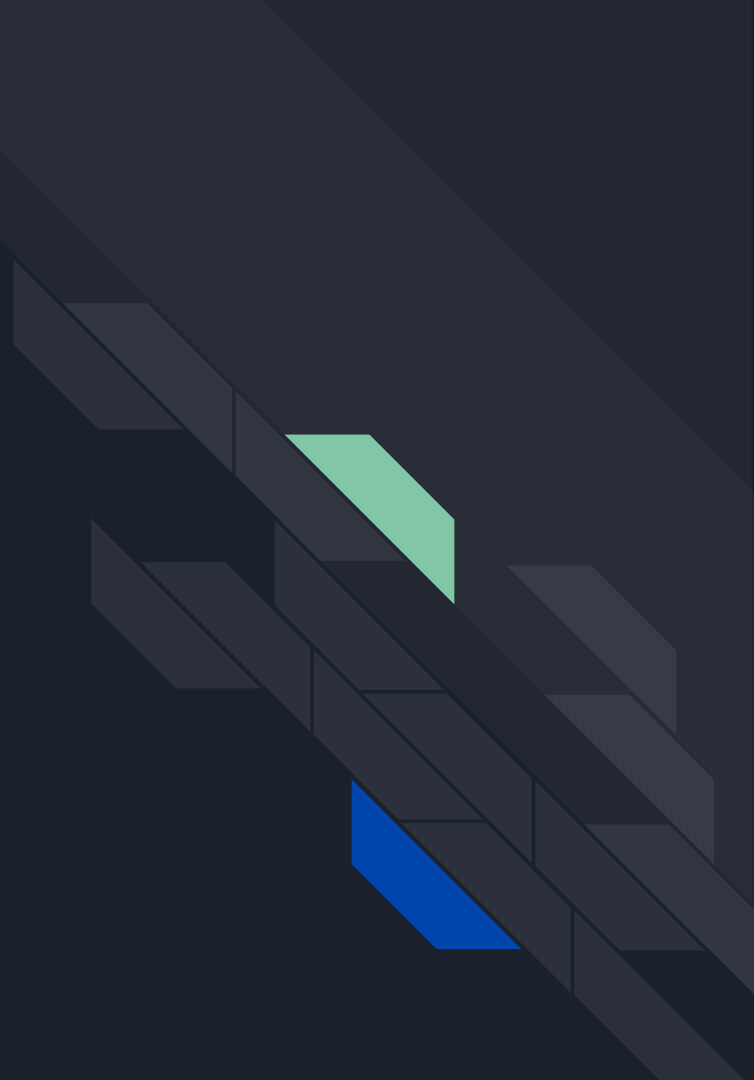
ExampleTest.php

```
<?php
```

```
it("has homepage")→skip();  
it("has contacts page")→skip();  
it("has about page")→skip();  
it("has registration")→skip();  
it("has login")→skip();
```



# Expectation API




# Assert multiple conditions

ExampleTest.php

```
<?php
```

```
test("expect things", function () {  
    $value = "Tech & Beer #2";  
  
    $this->assertSame($value, "Tech & Beer #2");  
    $this->assertIsString($value);  
});
```



# Assert multiple conditions with expect() function

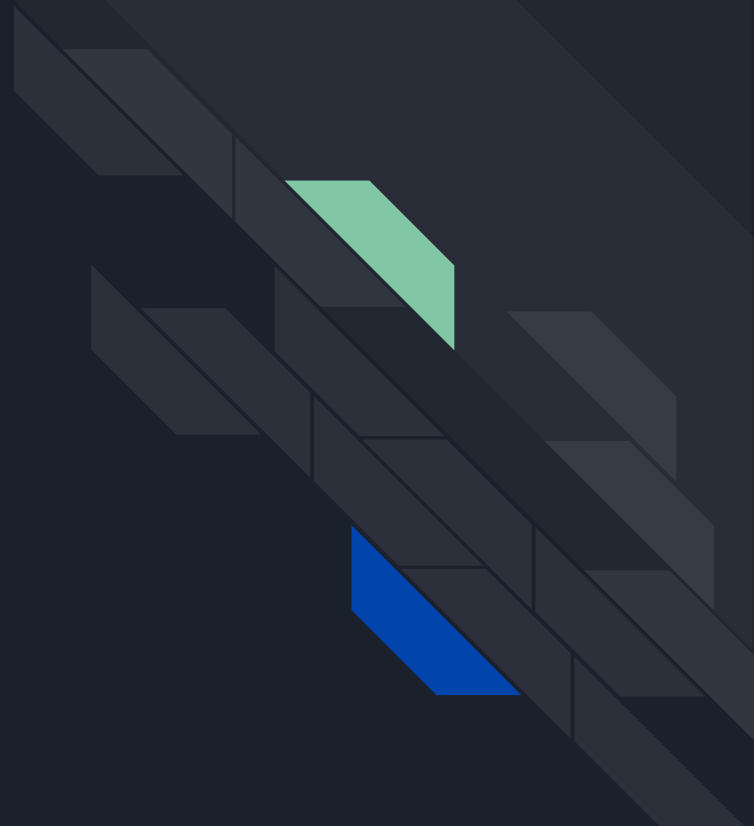
```
ExampleTest.php

<?php

test("expect things", function () {
    $value = "Tech & Beer #2";

    expect($value)
        →toBe("Tech & Beer #2")
        →toBeString()
        →not→toBe("Tech & Wine #2")
        →toContain("Beer");
});
```

# Datasets



# Using datasets

ExampleTest.php

```
<?php
```

```
test("has beers", function (string $beer) {  
    expect($beer)→not→toBeEmpty();  
})→with(["Kalopsia", "Glitch", "Sex Pills", "Acqua Santa"]);
```



# Using datasets

**PASS** Tests\Feature\ExampleTest

✓ has beers with ('Kalopsia')	0.05s
✓ has beers with ('Glitch')	0.01s
✓ has beers with ('Sex Pills')	0.01s
✓ has beers with ('Acqua Santa')	0.01s

Tests: **4 passed** (4 assertions)

Duration: **0.10s**

# Using datasets

ExampleTest.php

```
<?php
```

```
test("has drinks", function (array $drinks) {  
    expect($drinks)  
        →toBeArray()  
        →each→toBeString();  
})→with([  
    ["beers" ⇒ ["Kalopsia", "Glitch", "Sex Pills", "Acqua Santa"]],  
    ["cocktails" ⇒ ["Mojito", "Spritz", "Negroni"]],  
    ["soft" ⇒ ["Water", "Coke"]],  
]);
```



# Using datasets

**PASS** Tests\Feature\ExampleTest

- ✓ has drinks with (['Kalopsia', 'Glitch', 'Sex Pills', ...]) 0.04s
- ✓ has drinks with (['Mojito', 'Spritz', 'Negroni']) 0.01s
- ✓ has drinks with (['Water', 'Coke']) 0.01s

Tests: **3 passed** (12 assertions)

Duration: 0.09s



# Bound Datasets

ExampleTest.php

```
<?php
```

```
test("can generate email of a user", function (\App\Models\User $user)
{
    expect($user->email)
        ->not->toBeEmpty()
        ->toBeString();
})->with([fn() => \App\Models\User::factory()->create()]);
```



# Bound Datasets

**PASS** Tests\Feature\ExampleTest

✓ can generate email of a user with (Closure Object (...)) 0.08s

Tests: **1 passed** (2 assertions)

Duration: **0.11s**

# Sharing Datasets

- ▼ tests
  - ▼ Datasets
    - Beers.php
  - ▼ Feature
    - ExampleTest.php
  - ▼ Unit
    - ExampleTest.php
    - CreatesApplication.php
    - Pest.php
    - TestCase.php



# Sharing Datasets

 Datasets/Beers.php

```
<?php
```

```
dataset("beers", ["Kalopsia", "Glitch", "Sex Pills", "Acqua Santa"]);
```



# Sharing Datasets

ExampleTest.php

```
<?php
```

```
test("has beers (with shared dataset)", function (string $beer)
{
    expect($beer)→not→toBeEmpty();
})→with("beers");
```



# Sharing Datasets

**PASS** Tests\Feature\ExampleTest

- ✓ has beers (with shared dataset) with ('Kalopsia') 0.04s
- ✓ has beers (with shared dataset) with ('Glitch') 0.01s
- ✓ has beers (with shared dataset) with ('Sex Pills') 0.01s
- ✓ has beers (with shared dataset) with ('Acqua Santa') 0.01s

Tests: **4 passed** (4 assertions)

Duration: 0.10s

# Database



# Using database

ExampleTest.php

```
<?php
```

```
test("has users", function () {  
    \App\Models\User::factory()→create();  
  
    $this→assertDatabaseHas("users", [  
        "id" ⇒ 1,  
    ]);  
});
```





# Using database

**PASS** Tests\Feature\ExampleTest

✓ has users

0.07s

Tests: **1 passed** (1 assertions)

Duration: 0.10s

# Using database

ExampleTest.php

```
<?php

test("has users", function () {
    \App\Models\User::factory()→create();

    $this→assertDatabaseHas("users", [
        "id" ⇒ 1,
    ]);
});

test("has users 2", function () {
    \App\Models\User::factory()→create();

    $this→assertDatabaseHas("users", [
        "id" ⇒ 1,
    ]);
});
```

# Using database

ExampleTest.php

```
<?php

beforeEach(function () {
    \App\Models\User::factory()→create();
});

test("has users", function () {
    $this→assertDatabaseHas("users", [
        "id" ⇒ 1,
    ]);
});

test("has users 2", function () {
    $this→assertDatabaseHas("users", [
        "id" ⇒ 1,
    ]);
});
```

# Demo



We will integrate Pest on a simple Laravel project that expose some APIs.

<https://github.com/thebatclaudio/bat-tasks>



# Testing API: example

**POST** /api/tasks

Payload:

```
{  
  "title": "New Task",  
  "description": "Task description",  
  "status": "Planned"  
}
```

**ONLY ADMIN CAN CREATE TASKS!**

# Testing API: example

ExampleTest.php

```
<?php
```

```
test("admin can create tasks", function () {
    $user = \App\Models\User::factory()→create([
        "name" ⇒ "Batman"
    ]);

    $response = test()→actingAs($user)
        →post("/api/tasks", [
            "title" ⇒ "Fight Joker",
            "description" ⇒ "Fight Joker in ACE Chemicals",
            "status" ⇒ \App\Models\Task::STATUS["PLANNED"]
        ]);
    $response→assertStatus(201);
});
```

# Testing API: example

**FAIL** Tests\Feature\ExampleTest

✖ admin can create tasks

0.09s

---

**FAILED** Tests\Feature\ExampleTest > admin can create tasks

Expected response status code [201] but received 403.

Failed asserting that 201 is identical to 403.



# Testing API: example

ExampleTest.php

```
<?php
```

```
test("admin can create tasks", function () {
    $user = \App\Models\User::factory()→create([
        "name" ⇒ "Batman",
        "is_admin" ⇒ true,
    ]);

    $response = test()
        →actingAs($user)
        →post("/api/tasks", [
            "title" ⇒ "Fight Joker",
            "description" ⇒ "Fight Joker in ACE Chemicals",
            "status" ⇒ \App\Models\Task::STATUS["PLANNED"],
        ]);
    $response→assertStatus(201);
});
```





# Testing API: example

**PASS** Tests\Feature\ExampleTest

✓ admin can create tasks

0.08s

Tests: **1 passed** (1 assertions)

Duration: 0.12s



# Custom helpers

ExampleTest.php

```
<?php
```

```
function asAdmin(): \Tests\TestCase
{
    $user = \App\Models\User::factory()→create([
        "name" ⇒ "Batman",
        "is_admin" ⇒ true,
    ]);

    return test()→actingAs($user);
}
```

# Custom helpers

ExampleTest.php

```
<?php
```

```
test("admin can create tasks", function () {  
    $response = asAdmin()→post("/api/tasks", [  
        "title" ⇒ "Fight Joker",  
        "description" ⇒ "Fight Joker in ACE Chemicals",  
        "status" ⇒ \App\Models\Task::STATUS["PLANNED"],  
    ]);  
    $response→assertStatus(201);  
});
```



# Custom helpers

ExampleTest.php

```
<?php
```

```
function asSimpleUser(): \Tests\TestCase
{
    $user = \App\Models\User::factory()→create([
        "name" ⇒ "Robin",
        "is_admin" ⇒ false,
    ]);

    return test()→actingAs($user);
}
```

# Custom helpers

ExampleTest.php

```
<?php
```

```
test("user cannot create tasks", function () {  
    $response = asSimpleUser()→post("/api/tasks", [  
        "title" ⇒ "Fight Joker",  
        "description" ⇒ "Fight Joker in ACE Chemicals",  
        "status" ⇒ \App\Models\Task::STATUS["PLANNED"],  
    ]);  
    $response→assertStatus(403);  
});
```

# Check coverage (--coverage option)

```
./vendor/bin/pest --coverage
```

```
Tests:      1 skipped, 18 passed (31 assertions)
```

```
Duration: 0.54s
```

```
Console/Kernel ..... 19 / 66.7%
Exceptions/Handler ..... 56..58 / 70.0%
Http/Controllers/AuthController ..... 0.0%
Http/Controllers/Controller ..... 100.0%
Http/Controllers/TasksController ..... 15, 27..38 / 22.2%
Http/Kernel ..... 100.0%
Http/Middleware/Authenticate ..... 0.0%
Http/Middleware/EncryptCookies ..... 100.0%
Http/Middleware/PreventRequestsDuringMaintenance ..... 100.0%
```



## Check coverage (--min option)

```
./vendor/bin/pest --coverage --min=50
```

```
FAIL Code coverage below expected: 39.1 %. Minimum: 50.0 %.
```



# Parallel Testing

```
./vendor/bin/pest --parallel
```

```
..S.....
```

```
Tests:    1 skipped, 18 passed (31 assertions)
```

```
Duration: 0.46s
```

```
Parallel: 16 processes
```





# Parallel Testing


```
./vendor/bin/pest --parallel --processes=20
```

```
..S.....
```

```
Tests:    1 skipped, 18 passed (31 assertions)
```

```
Duration: 0.46s
```

```
Parallel: 16 processes
```



In conclusion,  
Pest:

- is easy to use
- is easy to learn
- is easy to read
- make easy to migrate from PHPUnit
- has a growing community and frequent updates



## Some links:

- <https://pestphp.com/>
- <https://github.com/pestphp/pest>
- <https://www.youtube.com/watch?v=MqiGA34ZrQU>
- <https://laravel.com/docs/10.x/readme>

Thank you!

