Testing Laravel Applications

Backend Brownbag Session | 08.02.2023

Agenda

- Types of tests
- PhpStorm Configuration
- DatabaseTransactions && RefreshDatabase
- Data Providers
- UrlGenerator
- Testing Events
- Testing Notifications
- Testing Files
- Mocking behavior
- Parallel running
- Memory issues with GitHub Actions

Types of tests

- low level - called **unit tests** - which test small pieces of code
- higher level tests such as **feature and integration tests** - which check that different parts of your code work well together, and
- **browser tests** - which interact with your code the same way a user would.

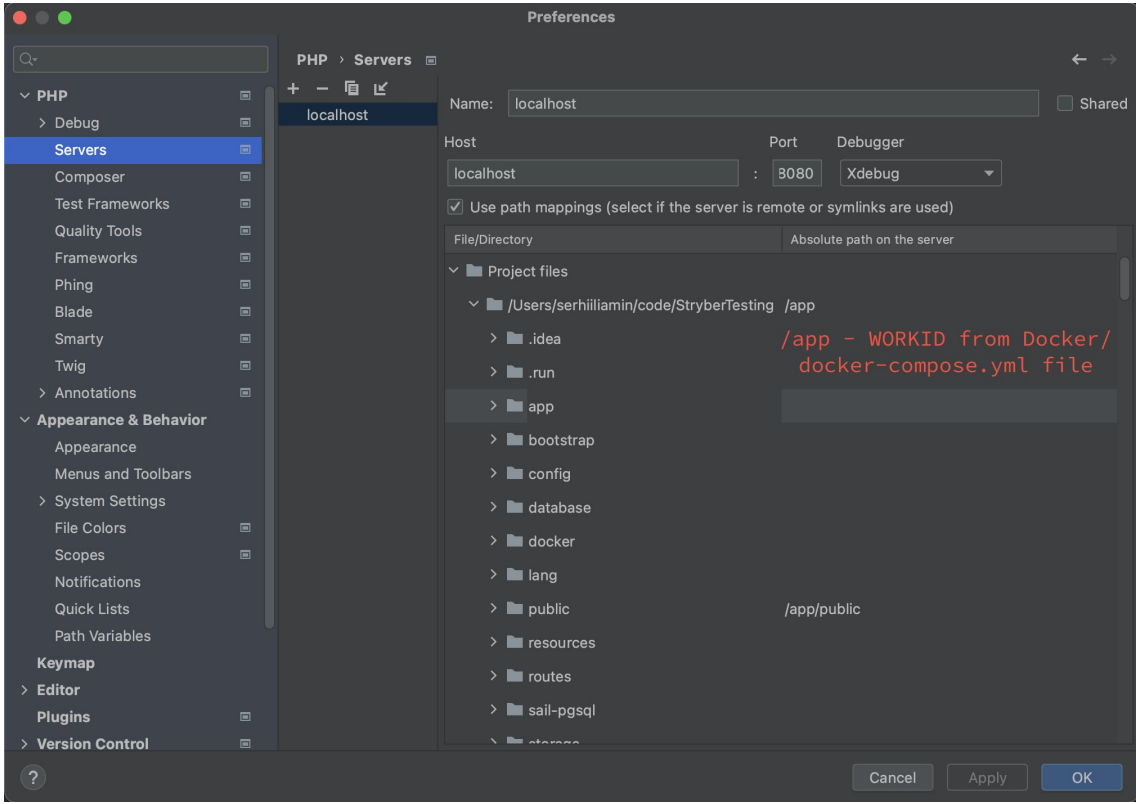
Each type of test focuses on the code differently. There are varied opinions on what each type of test should do and not do.

PhpStorm Configuration Pros:

- Running tests directly from PhpStorm
- Debugging code during test running
- Easy to filter passed/failsed tests
- Easy to wach output logs
- Possibility to use another database than described in **phpunit.xml**
- Save a lot of time during writing tests

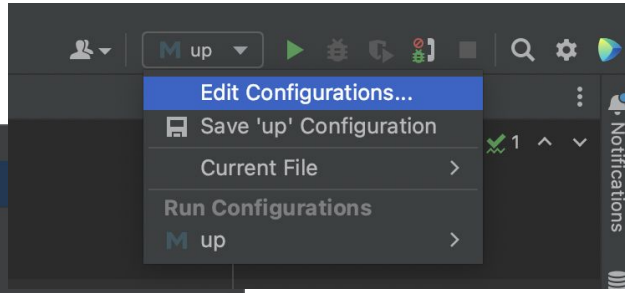
PhpStorm Configuration

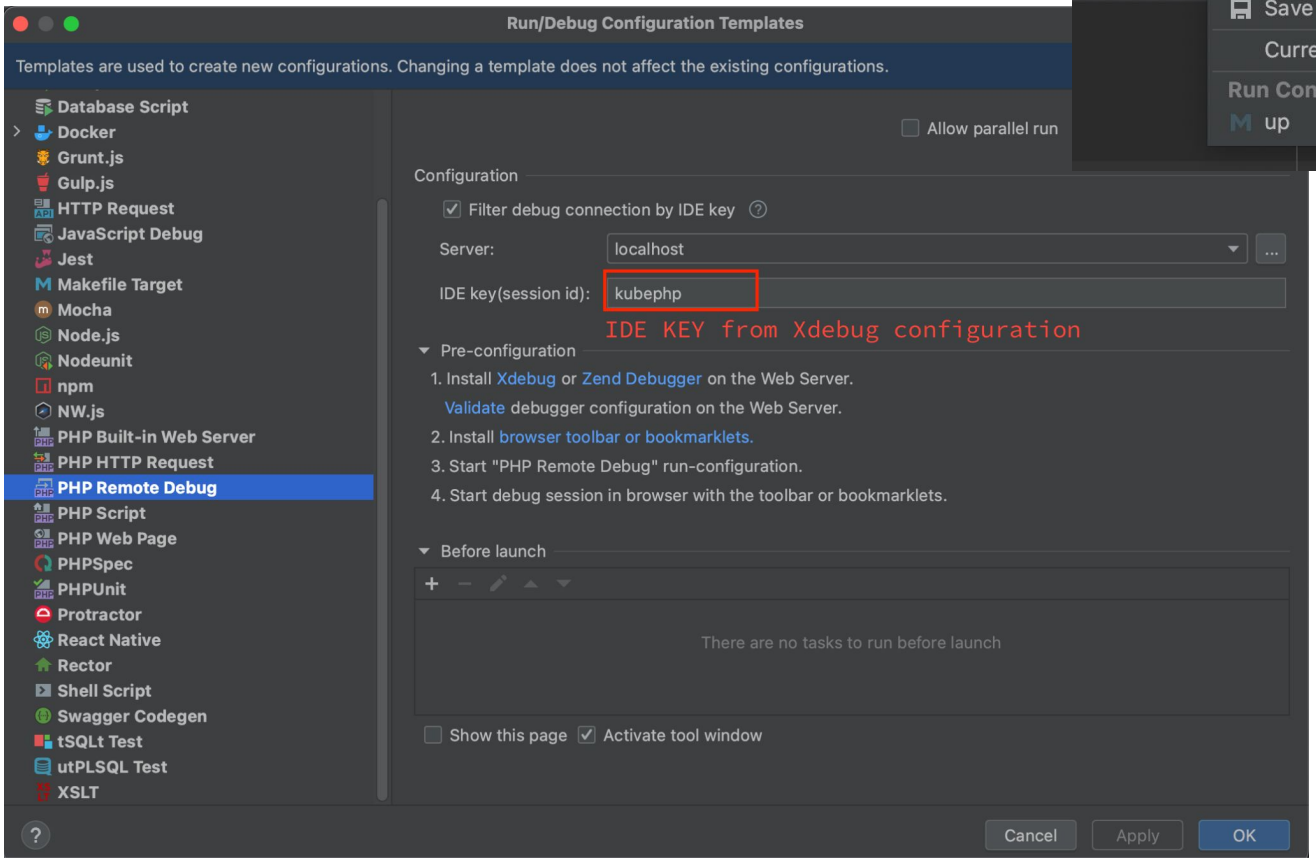
- Configure Xdebug



PhpStorm Configuration

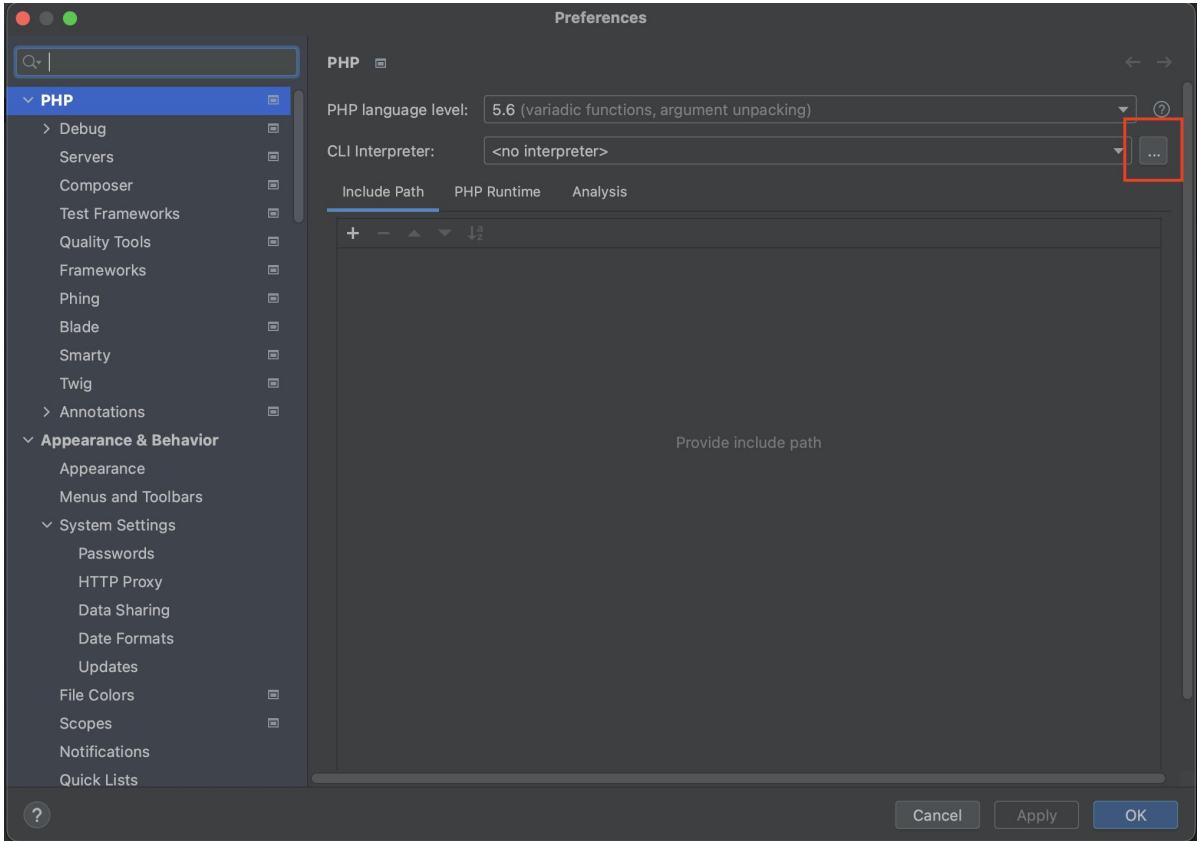
- Configure remote debug





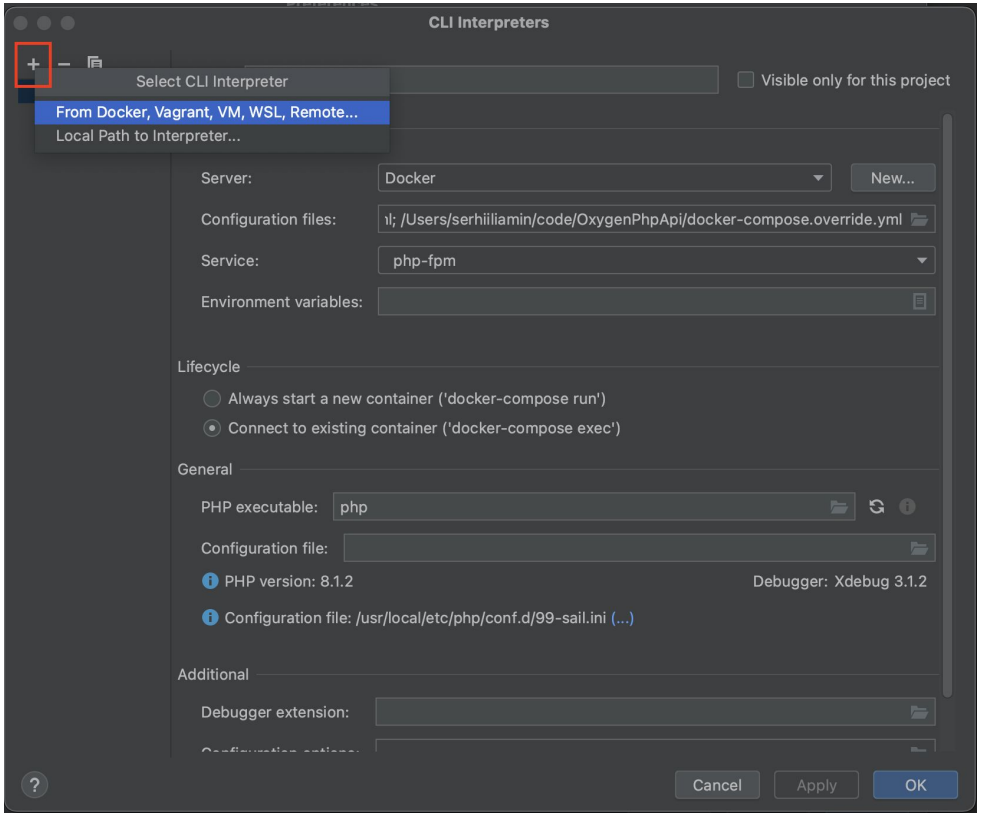
PhpStorm Configuration

- Configure CLI interpreter



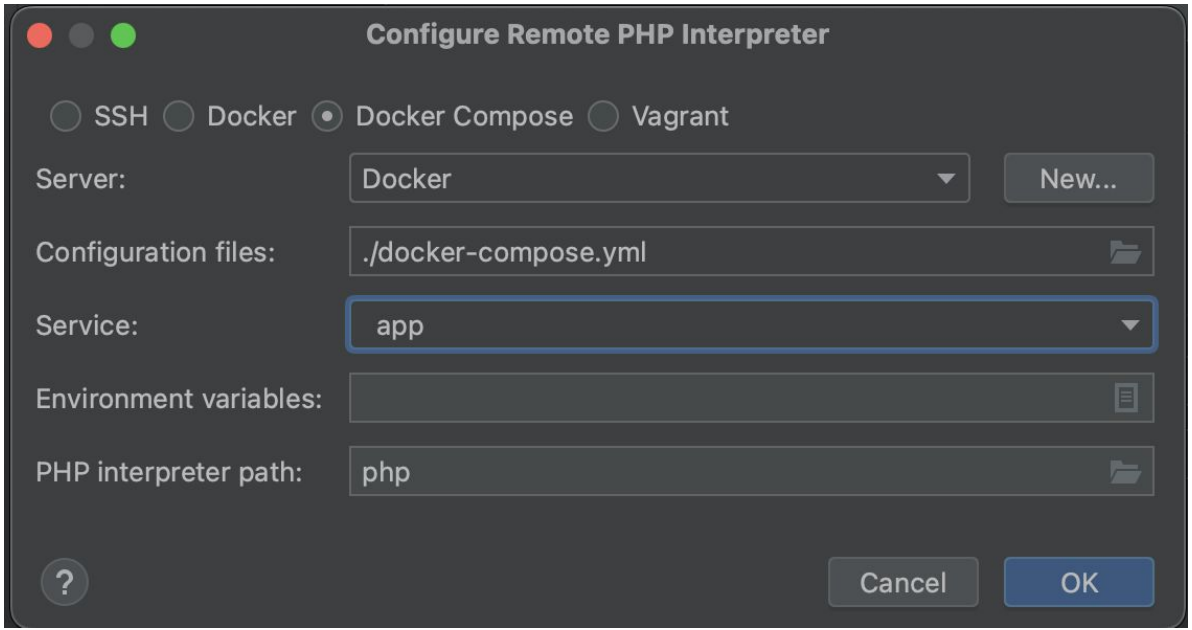
PhpStorm Configuration

- Configure CLI interpreter



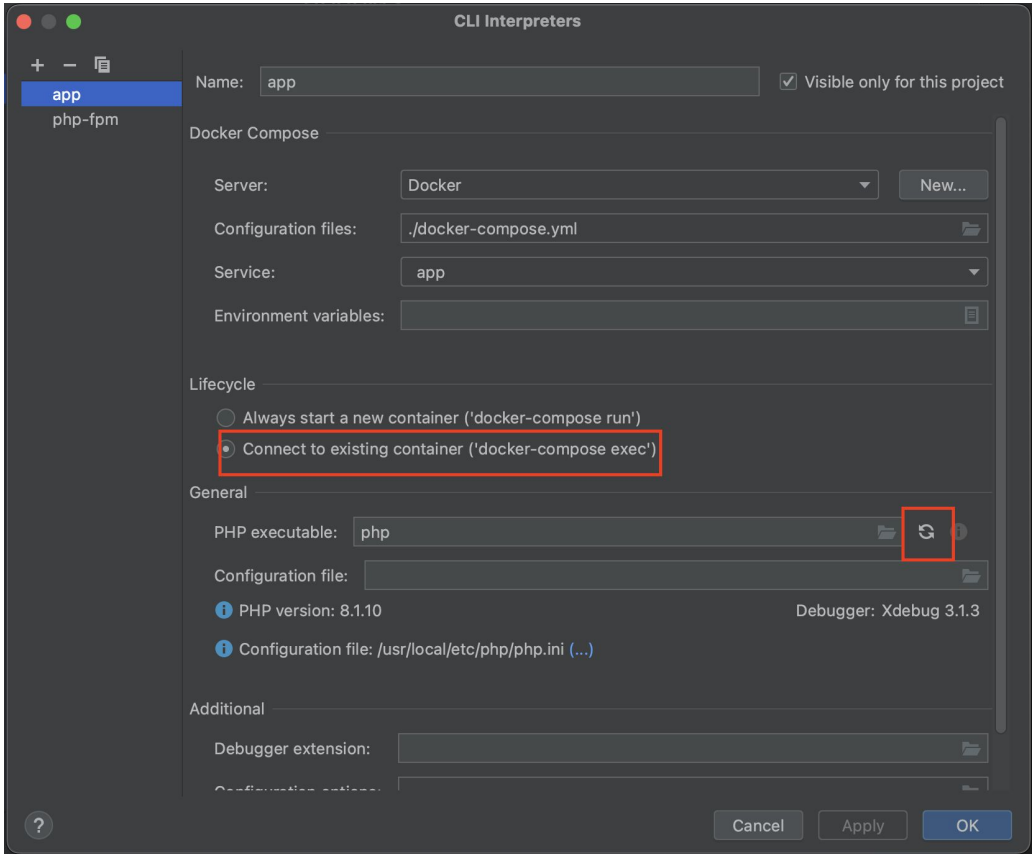
PhpStorm Configuration

- Configure CLI interpreter



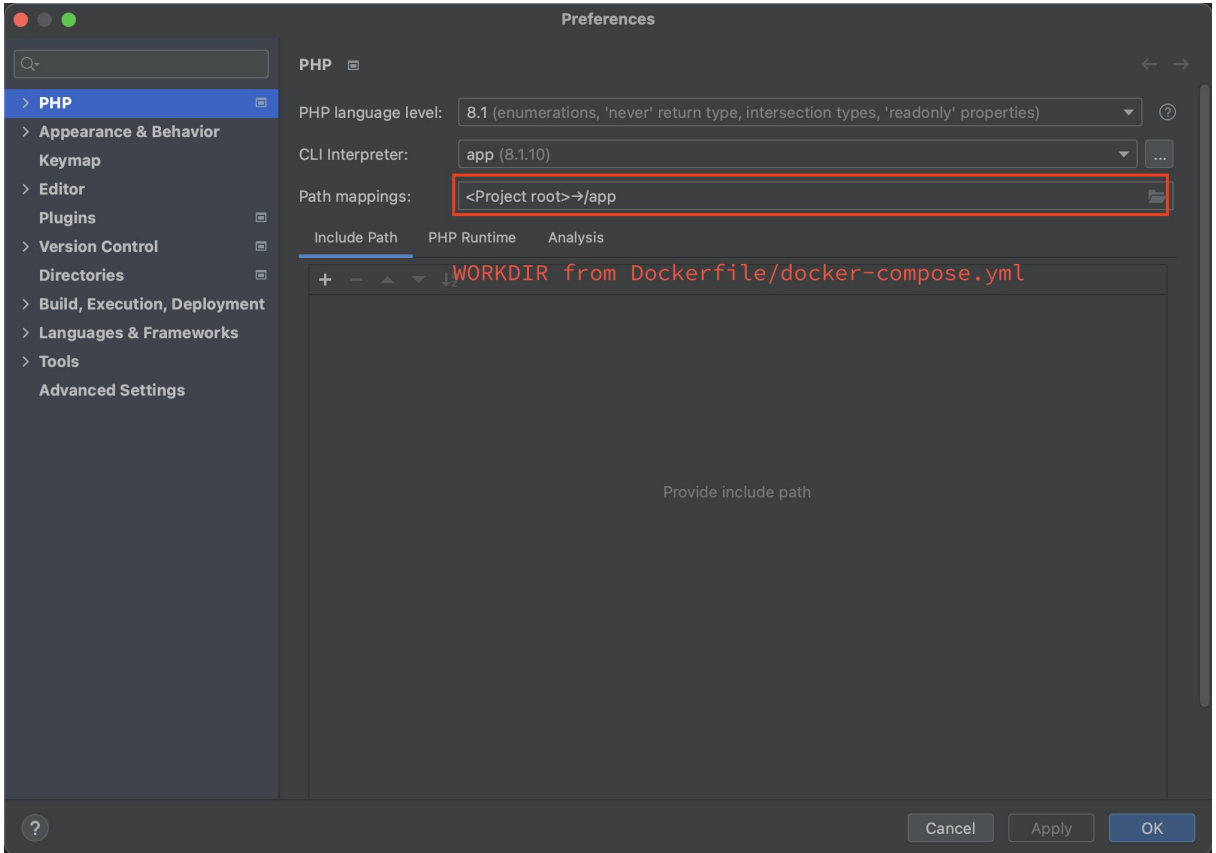
PhpStorm Configuration

- Configure CLI interpreter



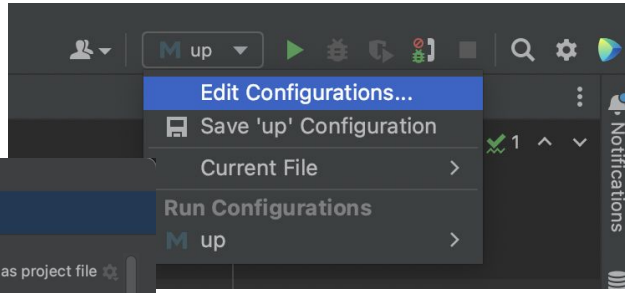
PhpStorm Configuration

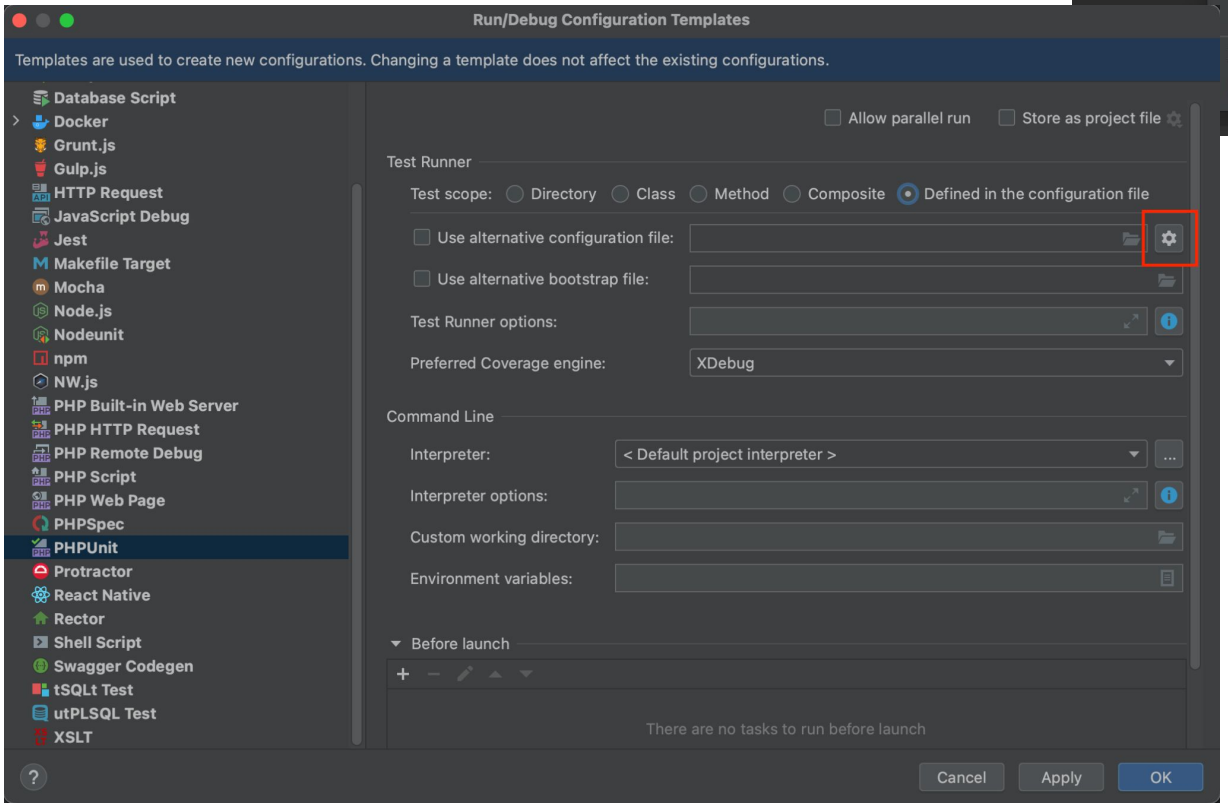
- Configure CLI interpreter



PhpStorm Configuration

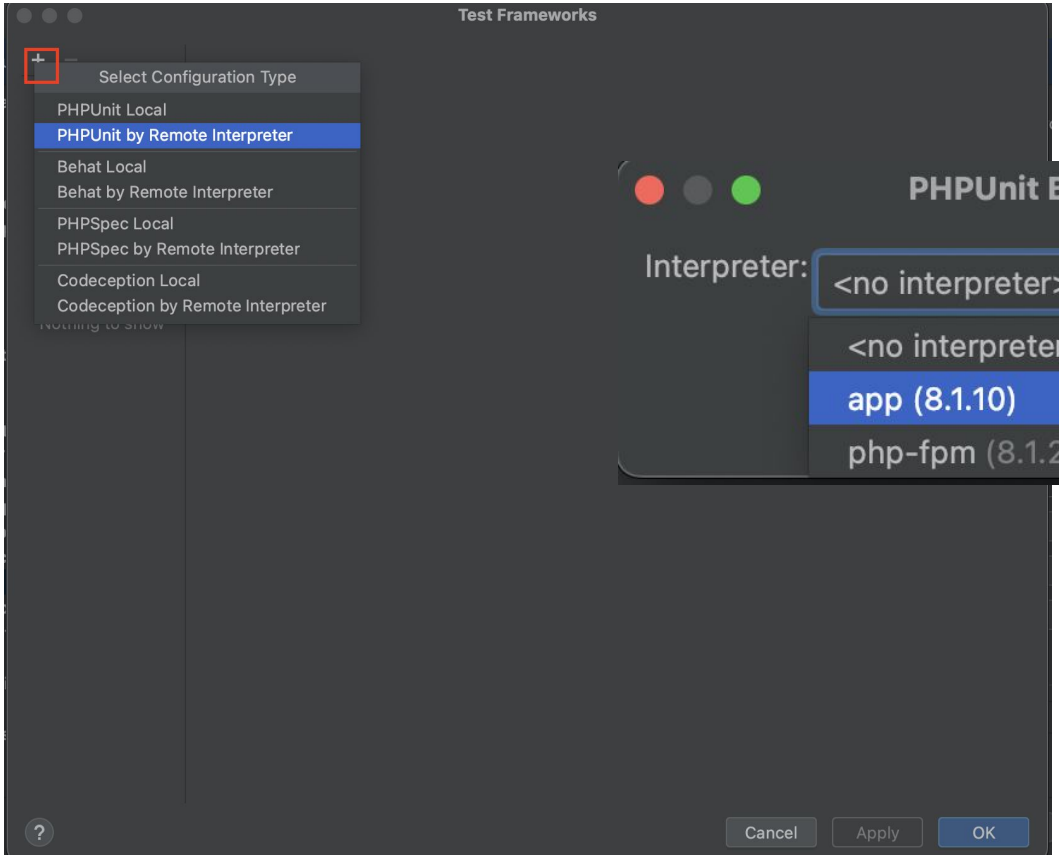
- Configure PhpUnit

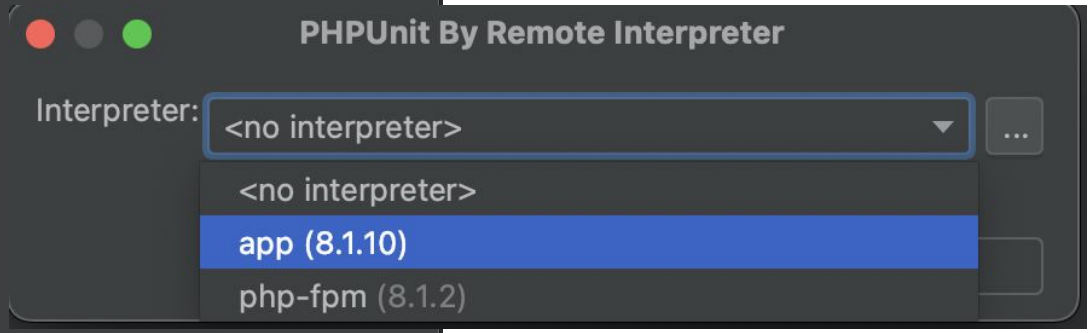




PhpStorm Configuration

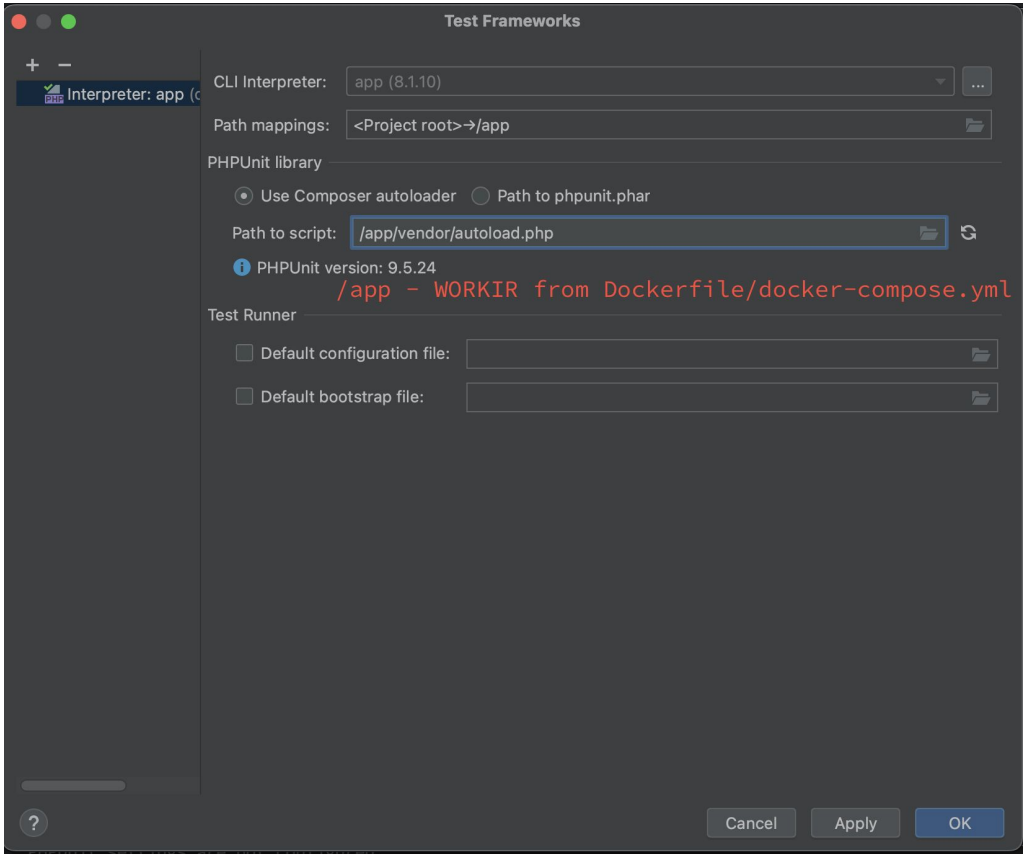
- Configure PhpUnit





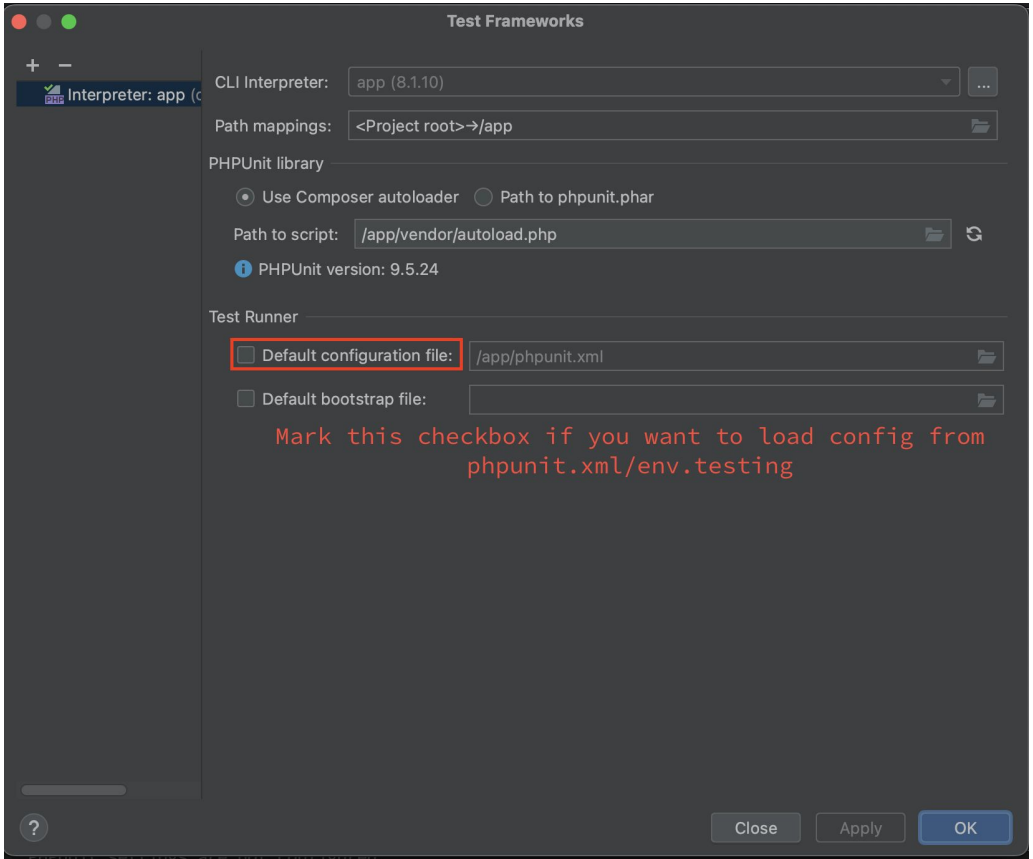
PhpStorm Configuration

- Configure PhpUnit



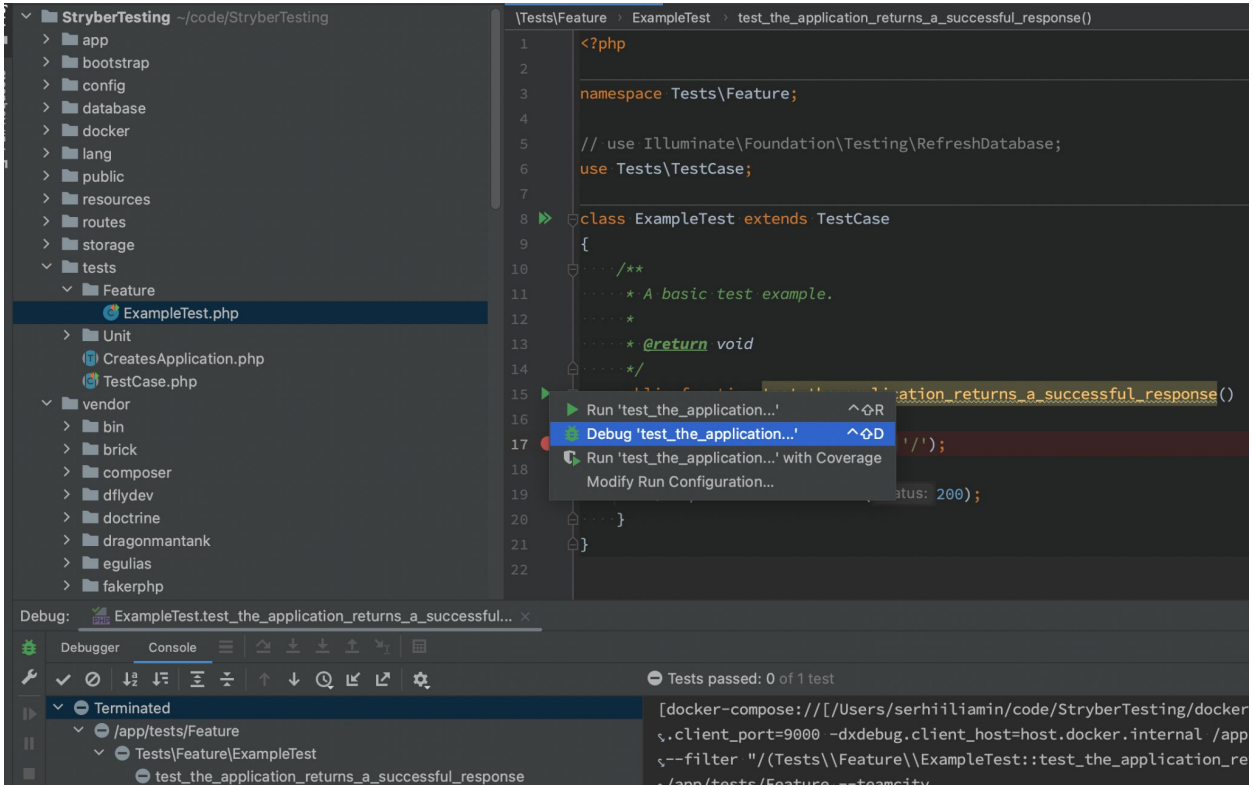
PhpStorm Configuration

- Configure PhpUnit



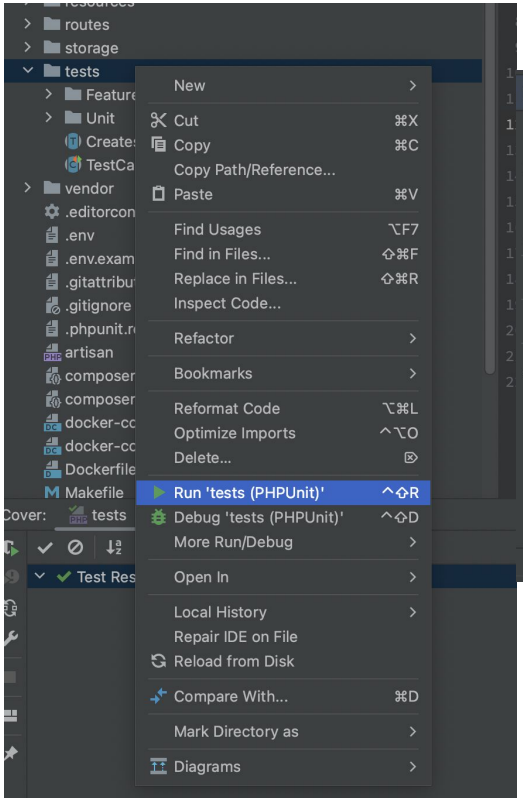
PhpStorm Configuration

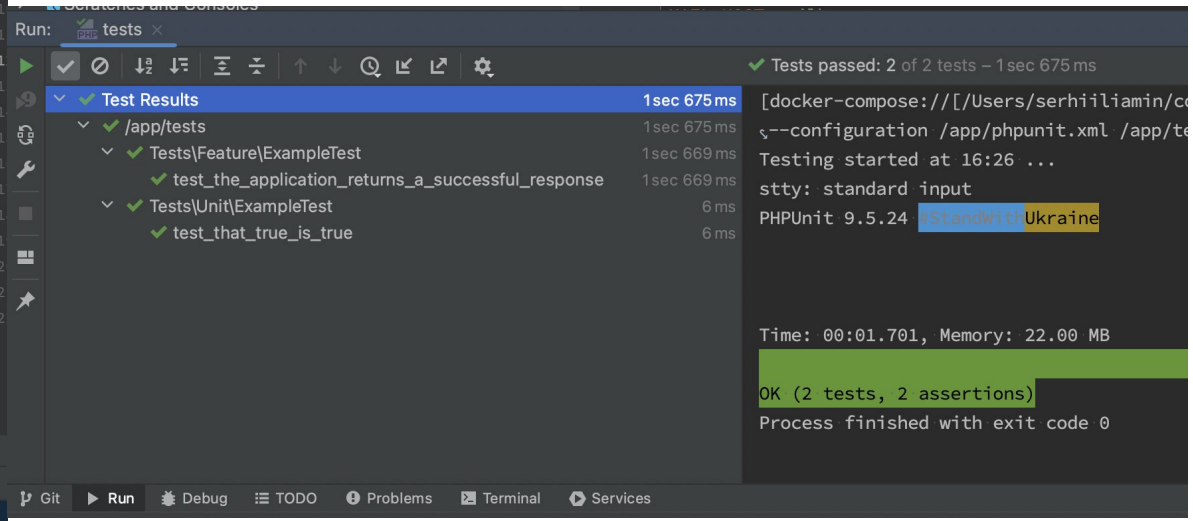
- After it you'll be able to run/debug tests directly from PhpStorm



PhpStorm Configuration

- After it you'll be able to run/debug tests directly from PhpStorm





DatabaseTransactions && RefreshDatabase

Laravel provides the **RefreshDatabase** trait **to reset the database after each test** so that data from a previous test does not interfere with subsequent tests.

The **RefreshDatabase** trait uses the artisan command **migrate:fresh** to drop all the tables from the database and then execute the **migrate** command

`DatabaseTransactions && RefreshDatabase`

In large applications with hundreds of migrations, the **migrate:fresh** could potentially slow down your tests.

DatabaseTransactions && RefreshDatabase

DatabaseTransactions, on the other hand, expects your database to be properly **migrated before your tests start**.

Then, it wraps every test in a database transaction, which it rolls back at the end of each test. This means that, at the end of each test, your database will be returned to the exact same state it was in prior to the test.

Data Providers

Data Providers are a handy feature of PHPUnit which allows you to **run the same test with different inputs** and expected results.

A data provider method must be **public** and either **return an array of arrays** or an **object that implements the *Iterator*** interface and yields an array for each iteration step.

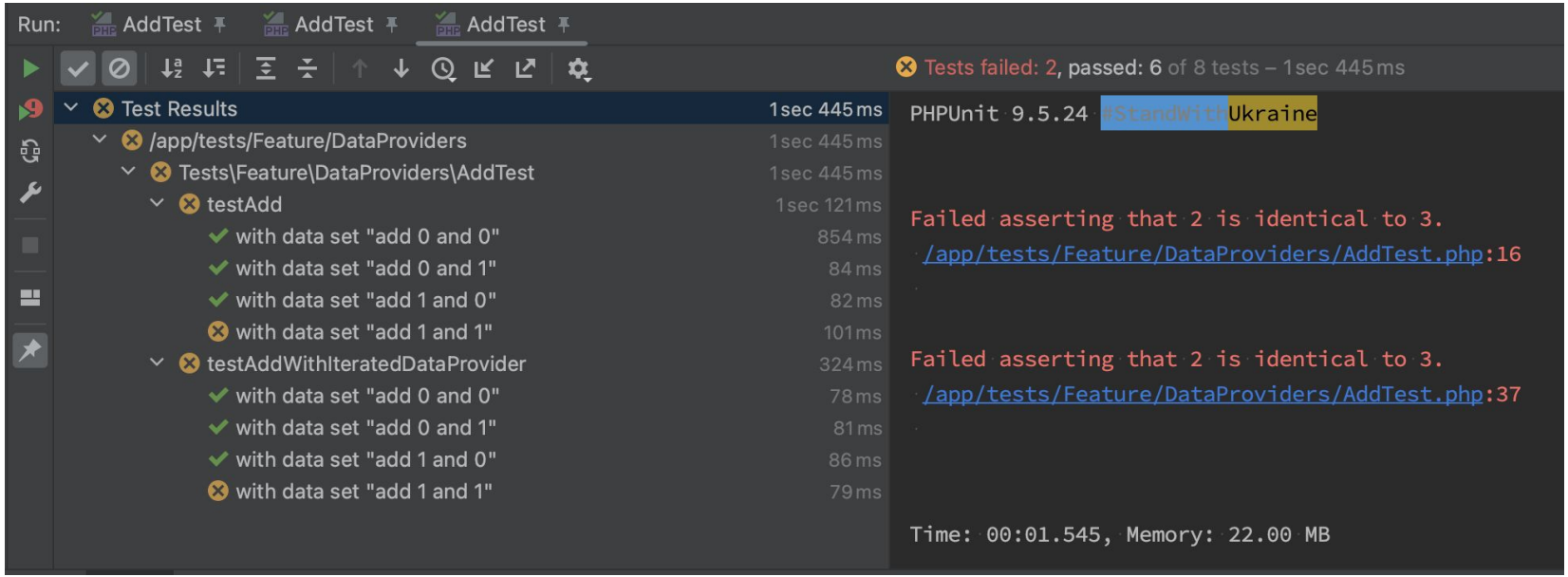
Data Providers

```php
/**
 * @dataProvider providerAddSuccess
 */
public function testAdd(int $a, int $b, int $expected): void
{
    $this->assertSame($expected, actual: $a + $b);
}

public function providerAddSuccess(): array
{
    return [
        /* [$firstArgument, $secondArgument, $expectedResult] */
        'add 0 and 0' => [0, 0, 0],
        'add 0 and 1' => [0, 1, 1],
        'add 1 and 0' => [1, 0, 1],
        'add 1 and 1' => [1, 1, 3]
    ];
}
```

Data Providers

```
32      /**
33       * @dataProvider providerAddSuccessIterated
34       */
35      public function testAddWithIteratedDataProvider(int $a, int $b, int $expected): void
36      {
37          $this->assertSame($expected, actual: $a + $b);
38      }
39
40      public function providerAddSuccessIterated(): iterable
41      {
42          /* [$firstArgument, $secondArgument, $expectedResult] */
43          yield 'add 0 and 0' => [0, 0, 0];
44          yield 'add 0 and 1' => [0, 1, 1];
45          yield 'add 1 and 0' => [1, 0, 1];
46          yield 'add 1 and 1' => [1, 1, 3];
47      }
```

Data Providers



Data Providers Tips:

- Always **name** the data providers.
- Add **type definitions**.
- You can use Closures in Data providers to delay evaluation.
- Use **yield** to simplify large nested arrays
- The **disadvantage** of data providers is that they are **evaluated before anything else** (to allow PHPUnit calculate the total number of tests). It means that they can't access anything initialized in setUpBeforeClass() or setUp().

Data Providers and access to the Test methods/properties

```php
/**
 * @dataProvider providerAddSuccessClosure
 */
public function testAddWithClosureDataProvider(string $expectedResult, Closure $setTime): void
{
    $dateTime = new DateTime( datetime: '2019-09-01');

    // pass reference to the current class to have access to the class properties/methods
    $setTime($this, $dateTime);

    self::assertSame($expectedResult, $dateTime->format( format: 'Y-m-d H:i:s'));
}
```

Data Providers and access to the Test methods/properties

```php
// Data Provider with closure
public function providerAddSuccessClosure(): iterable
{
    yield 'midnight' => [
        '2019-09-01 00:00:00',
        function (AddTest $self, DateTime $date): void {
            // do something with current test instance
            // $self->user->id

            $date->setTime( hour: 0, minute: 0, second: 0);
        },
    ];
}
```

UrlGenerator

```
Route::get( uri: '/some-endpoint/{id}', action: 'App\Http\Actions\Some\SomeAction')
    ->where( name: 'id', expression: '[0-9]+')
    ->name( name: 'some-endpoint')
;
```

UrlGenerator allow us to get API endpoint URL based on it's name/action

UrlGenerator

```php
final class SomeEndpointTest extends TestCase
{
    public function testSomeEndpointSuccessResponse(): void
    {
        $number = $this->faker->randomNumber( nbDigits: 3);

        $urlByAction = $this->urlGenerator->action( action: SomeAction::class, [
            'id' => $number
        ]);

        $urlByName = $this->urlGenerator->route( name: 'some-endpoint', [
            'id' => $number
        ]);

        $this->assertEquals($urlByAction, $urlByName);

        $this->getJson($urlByAction)
            ->assertOk()
            ->assertExactJson([
                JsonResource::$wrap => [
                    $number
                ],
            ])
        ;
    }
}
```

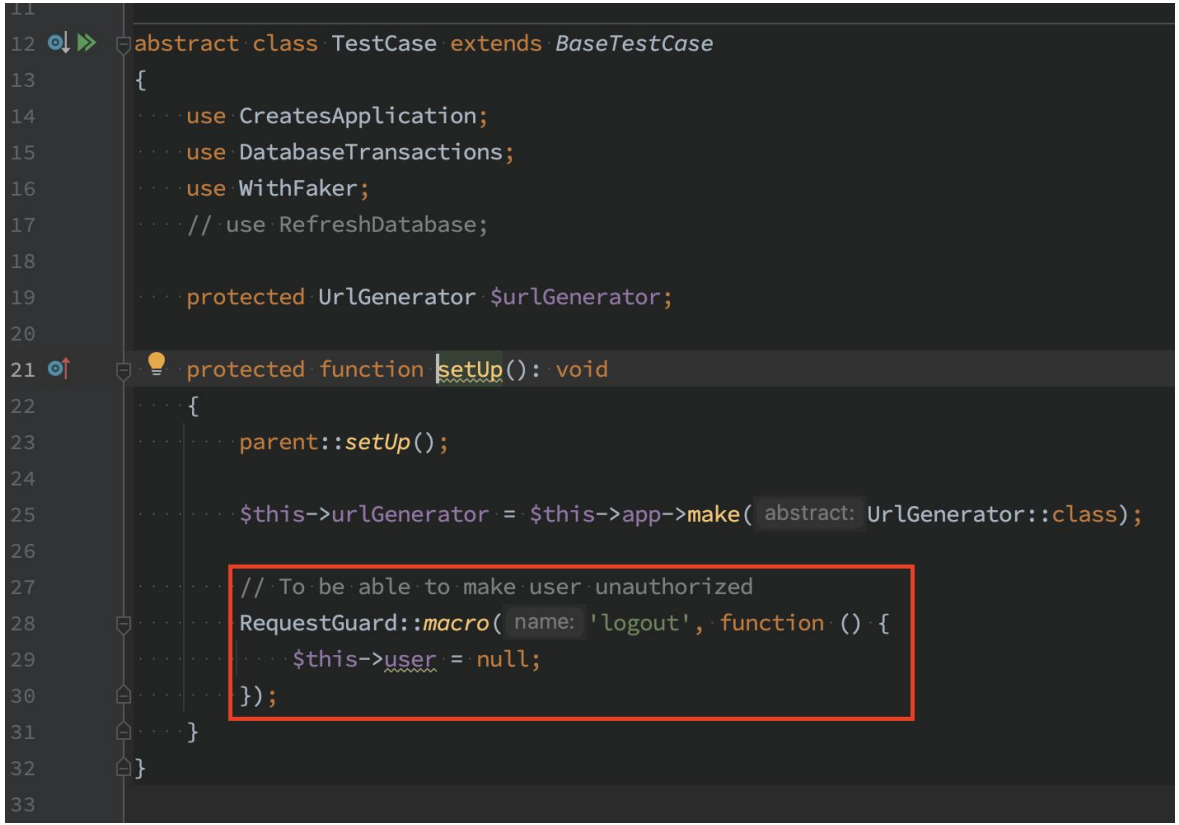
Authentication what to check:

- Always check that API endpoint isn’t accesible fro unauthorized users
- Authentication flow (log in/log out)
- HTTP **forbidden** code **403**

Authentication

```php
trait WithAuthentication
{
    final protected function authAsUser(?User $user = null): User
    {
        if (empty($user)) {
            $user = User::factory()->create();
        }

        $this->actingAs($user, guard: 'sanctum');

        return $user;
    }

    protected function logout(): void
    {
        /** @uses \Tests\TestCase::setUp() */
        Auth::guard( name: 'sanctum')->logout();
    }
}
```

Authentication



```php
abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;
    use DatabaseTransactions;
    use WithFaker;
    // use RefreshDatabase;

    protected UrlGenerator $urlGenerator;

    protected function setUp(): void
    {
        parent::setUp();

        $this->urlGenerator = $this->app->make( abstract: UrlGenerator::class);

        // To be able to make user unauthorized
        RequestGuard::macro( name: 'logout', function () {
            $this->user = null;
        });
    }
}
```

Testing Events what to check:

- Event is dispatched with parameters that we expect
- Listeners are listening for dispatched event
- Listeners are handling dispatched event in a proper way
- Listenres were/weren’t queued

Testing Events

```php
<?php

namespace App\Events;

use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

final class ApiEndpointCalledEvent
{
    use Dispatchable, SerializesModels;

    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct(private string $endpointName)
    {
    }

    public function getEndpointName(): string
    {
        return $this->endpointName;
    }
}
```

```php
<?php

namespace App\Listeners;

use App\Events\ApiEndpointCalledEvent;
use Illuminate\Contracts\Queue\ShouldQueue;
use Psr\Log\LoggerInterface;

final class ListenToApiEndpointCalled implements ShouldQueue
{
    /**
     * Create the event listener.
     *
     * @return void
     */
    public function __construct(private LoggerInterface $logger)
    {
    }

    /**
     * Handle the event.
     *
     * @param  \App\Events\ApiEndpointCalledEvent  $event
     *
     * @return void
     */
    public function handle(ApiEndpointCalledEvent $event): void
    {
        $this->logger->info($event->getEndpointName() . ' Endpoint called');
    }
}
```

Testing Events

```php
<?php

declare(strict_types=1);

namespace App\Http\Actions\ActionWithEvent;

use ...

class EventAction extends Controller
{
    public function __construct(
        private Dispatcher $eventDispatcher
    ) {
    }

    public function __invoke(): JsonResource
    {
        // event(new ApiEndpointCalledEvent(Route::currentRouteName()));
        // ApiEndpointCalledEvent::dispatch(Route::currentRouteName());
        $this->eventDispatcher->dispatch(
            new ApiEndpointCalledEvent(Route::currentRouteName())
        );

        return new JsonResource(['ok']);
    }
}
```

Testing Events

```php
public function testEventWasDispatched(): void
{
    $dispatcher = Event::fake([
        ApiEndpointCalledEvent::class,
    ]);

    $routeName = 'create-event';
    $url = $this->urlGenerator->route($routeName);
    $this->getJson($url)->assertOk();

    $dispatcher->assertDispatched( event: ApiEndpointCalledEvent::class);

    $dispatcher->assertDispatched(function (ApiEndpointCalledEvent $event) use ($routeName) {
        return $event->getEndpointName() === $routeName;
    });
}
```

Testing Events

```php
public function testListenerIsListening(): void
{
    $dispatcher = Event::fake([
        ApiEndpointCalledEvent::class,
    ]);

    $routeName = 'create-event';
    $url = $this->urlGenerator->route($routeName);
    $this->getJson($url)->assertOk();

    $dispatcher->assertListening(
        expectedEvent: ApiEndpointCalledEvent::class,
        [ListenToApiEndpointCalled::class, 'handle']
    );
}
```

Testing Events

```php
public function testEventListenerWasQueued(): void
{
    Queue::fake();

    $routeName = 'create-event';
    $url = $this->urlGenerator->route($routeName);
    $this->getJson($url)->assertOk();

    // assertNotPushed if listener doesn't implement Illuminate\Contracts\Queue\ShouldQueue
    Queue::assertPushed(
        job: CallQueuedListener::class,
        function ($listener) {
            return $listener->class === ListenToApiEndpointCalled::class;
        }
    );
}
```

Testing Events

```php
    public function testEventListenerWork(): void
    {
        /** @var MockInterface|LoggerInterface $listener */
        $logger = $this->partialMock( abstract: LoggerInterface::class);

        $routeName = 'create-event';

        $logger
            ->shouldReceive( ...methodNames: 'info')
            ->with( ...args: $routeName . ' Endpoint called')
            ->once()
        ;

        $url = $this->urlGenerator->route($routeName);
        $response = $this->getJson($url);
        $response->assertOk();
    }
}
```

Testing Events

```php
public function testEventListenerWorkOtherWay(): void
{
    /** @var MockInterface|LoggerInterface $listener */
    $logger = $this->partialMock( abstract: LoggerInterface::class);

    $routeName = 'create-event';

    $logger
        ->shouldReceive( ...methodNames: 'info')
        ->with( ...args: $routeName . ' Endpoint called')
        ->once()
    ;

    (new ListenToApiEndpointCalled($logger))->handle(new ApiEndpointCalledEvent($routeName));
}
}
```

Exceptions what to check:

- Response HTTP code
- Error response message

For custom API exceptions:

- Error response code
- Error response fallback_message
- Error response payload

Checking exceptions

```
11  class ExceptionAction extends Controller
12  {
13      public function __invoke(): JsonResource
14      {
15          throw new SomeException();
16      }
17  }
18
```

Checking exceptions

```php
final class SomeException extends DomainException implements ApiException
{
    public const CODE = 'exception.code.for.frontend';

    public function getExceptionCode(): string
    {
        return self::CODE;
    }

    public function getFallbackMessage(): string
    {
        return __( key: 'exceptions.group.message');
    }

    public function getPayload(): array
    {
        return [];
    }

    public function getHttpCode(): int
    {
        return Response::HTTP_UNPROCESSABLE_ENTITY;
    }
}
```

Checking exceptions

```php
40      /**
41       * Register the exception handling callbacks for the application.
42       *
43       * @return void
44       */
45      public function register()
46      {
47          $this->reportable(function (Throwable $e) {
48              //
49          });
50          $this->renderable(function (ApiException $e, $request) {
51              return (new ApiExceptionResource($e))->toResponse($request);
52          });
53      }
```

Checking exceptions

```php
/** @mixin ApiException */
final class ApiExceptionResource extends JsonResource
{
    public static $wrap = 'error';

    public const KEY_CODE = 'code';
    public const KEY_FALLBACK_MESSAGE = 'fallback_message';
    public const KEY_PAYLOAD = 'payload';

    public function toArray($request): array
    {
        return [
            self::KEY_CODE => $this->getExceptionCode(),
            self::KEY_FALLBACK_MESSAGE => $this->getFallbackMessage(),
            self::KEY_PAYLOAD => $this->getPayload(),
        ];
    }

    public function withResponse($request, $response): void
    {
        $response->setStatusCode($this->getHttpCode());
    }
}
```

Checking exceptions

```php
final class ExceptionsTest extends TestCase
{
    public function testExceptionIsThrown(): void
    {
        $url = $this->urlGenerator->route( name: 'make-exception');

        $response = $this->getJson($url);

        // Exceptions that don't inherit from App\Exceptions\ApiException\ApiException::class
        // $this->assertStringContainsString(
        //     __('exceptions.group.message'),
        //     $response->json('message')
        // );

        // Exceptions that inherit from App\Exceptions\ApiException\ApiException::class
        $response
            ->assertStatus( status: Response::HTTP_UNPROCESSABLE_ENTITY)
            ->assertJson(fn(AssertableJson $json) =>
                $json->has(
                    'error',
                    fn(AssertableJson $json) => $json
                        ->where(
                            key: ApiExceptionResource::KEY_CODE,
                            expected: SomeException::CODE
                        )
                        ->where( key: ApiExceptionResource::KEY_FALLBACK_MESSAGE, __( key: 'exceptions.group.message'))
                        ->where( key: ApiExceptionResource::KEY_PAYLOAD, [])
                        ->etc()
                )
            )
        ;
    }
}
```

Validation what to check:

- Response HTTP code
- All required parameters
- Parameters values (min/max/before/after/in/exists/etc)
- Custom validation rules messages
- Validation messages with custom attributes

Testing Validation errors

```php
/**
 * @property-read string $pay_day
 * @property-read string $invoice_cutoff_days
 * @property-read string $IBAN
 * @property-read string $some_field
 */
final class ValidationActionRequest extends ApiRequest
{
    public const KEY_PAY_DAY = 'pay_day';
    public const KEY_INVOICE_CUTOFF_DAYS = 'invoice_cutoff_days';
    public const KEY_IBAN = 'IBAN';
    public const KEY_SOME_FIELD = 'some_field';

    function rules(): array
    {
        return [
            self::KEY_PAY_DAY => [
                'required',
                'integer',
                'min:1',
                'max:31',
            ],
            self::KEY_INVOICE_CUTOFF_DAYS => [
                'required',
                'integer',
                'min:' . User::MIN_CUTOFF_DAYS,
                'max:' . User::MAX_CUTOFF_DAYS,
                new InvoiceCutoffDaysConsideringPayDay((int) $this->invoice_cutoff_days, (int) $this->pay_day),
            ],
            self::KEY_IBAN => [
                'required',
                'string',
            ],
            self::KEY_SOME_FIELD => [
                'required',
                'string',
            ],
        ];
    }
}
```

Testing Validation errors

```php
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

final class InvoiceCutoffDaysConsideringPayDay implements Rule
{
    public function __construct(
        private int $invoiceCutoffDays,
        private int $payDay,
    ) {
    }

    public function passes($attribute, $value): bool
    {
        return $this->invoiceCutoffDays <= $this->payDay;
    }

    public function message(): string
    {
        return __( key: 'validation.custom.wrong_invoice_cutoff_days', [
            'pay_day' => $this->payDay,
        ]);
    }
}
```

Testing Validation errors

```
156     'custom' => [
157         'attribute-name' => [
158             'rule-name' => 'custom-message',
159         ],
160         'wrong_invoice_cutoff_days' => ''
161             . ValidationActionRequest::KEY_INVOICE_CUTOFF_DAYS
162             . '" should be less or equal than :pay_day if '
163             . '"'
164             . ValidationActionRequest::KEY_PAY_DAY
165             . '"'
166             . ' equals :pay_day'
167         ,
168     ],
169
170     /*
171     |--------------------------------------------------------------------------
172     | Custom Validation Attributes
173     |--------------------------------------------------------------------------
174     |
175     | The following language lines are used to swap our attribute placeholder
176     | with something more reader friendly such as "E-Mail Address" instead
177     | of "email". This simply helps us make our message more expressive.
178     |
179     */
180
181     'attributes' => [
182         'IBAN' => 'IBAN',
183     ],
```

Testing Validation errors

```php
    public function testValidationErrorIsThrown(): void
    {
        $payDay = $this->faker->numberBetween( int1: 1, int2: 30);
        $invoiceCutoffDays = $payDay + 1;

        $response = $this->getJson(
            $this->urlGenerator->action( action: ValidationAction::class, [
                ValidationActionRequest::KEY_PAY_DAY => $payDay,
                ValidationActionRequest::KEY_INVOICE_CUTOFF_DAYS => $invoiceCutoffDays,

                // Without required field
                // ValidationActionRequest::KEY_IBAN => $this->faker->iban,
                // ValidationActionRequest::KEY_SOME_FIELD => $this->faker->name,
            ])
        )->assertUnprocessable();

        $errors = $response->json( key: 'errors');

```

Testing Validation errors

```
32        // ================================================================================
33        // Check custom validation message
34
35        $this->assertArrayHasKey( key: ValidationActionRequest::KEY_INVOICE_CUTOFF_DAYS, $errors);
36        $this->assertContains(
37            __(
38                key: 'validation.custom.wrong_invoice_cutoff_days',
39                [
40                    'pay_day' => $payDay
41                ]
42            ),
43            $response->json( key: 'errors.' . ValidationActionRequest::KEY_INVOICE_CUTOFF_DAYS)
44        );
45
```

Testing Validation errors

```php
        // ================================================================================================
        // Check required field validation error with custom validation attribute

        /** @var Translator $translator */
        $translator = app( abstract: Translator::class);

        $requiredMissingFields = [
            ValidationActionRequest::KEY_IBAN,
            ValidationActionRequest::KEY_SOME_FIELD
        ];

        foreach ($requiredMissingFields as $requiredMissingField) {
            $this->assertArrayHasKey($requiredMissingField, $errors);
            $this->assertContains(
                __(
                    key: 'validation.required',
                    [
                        // 'The i b a n field is required.' => 'The IBAN field is required.'
                        // 'The some field field is required.'
                        'attribute' => Arr::get(
                            $translator->get( key: 'validation.attributes'),
                            $requiredMissingField
                        )
                        ?? str_replace( search: '_', replace: ' ', Str::snake($requiredMissingField))
                    ]
                ),
                $response->json( key: 'errors.' . $requiredMissingField)
            );
        }
    }
```

Notifications what to check:

- Notification was send
- Notification recipient
- Notification channels
- Notification **toArray()** parameters
- Notification mail attributes(subject/template)
- Notification mail content
- Notification was/wasn’t queued

Testing Notifications

```php
final class ApiEndpointCalledNotification extends Notification implements ShouldQueue
{
    use Queueable;

    public const SUBJECT = 'API endpoint called';
    public const TEMPLATE = 'emails.api-endpoint-called';

    public function __construct(
        private string $route
    ) {
        $this->queue = QUEUE_NOTIFICATIONS;
    }

    public function via(User $notifiable): array
    {
        return [MailChannel::class];
    }

    public function toMail(User $notifiable): MailMessage
    {
        return (new MailMessage())
            ->subject( subject: self::SUBJECT)
            ->markdown( view: self::TEMPLATE, $this->toArray($notifiable))
        ;
    }

    public function toArray(User $notifiable): array
    {
        return [
            'route' => $this->route,
        ];
    }
}
```

Testing Notifications

```php
final class NotificationAction extends Controller
{
    public function __invoke(): JsonResource
    {
        try {
            /** @var Notifiable $user */
            $user = User::query()->latest()->firstOrFail();
        } catch (ModelNotFoundException $e) {
            return new JsonResource([
                'no active users'
            ]);
        }

        try {
            // In the real life application it's better to create event and listener for sending email
            $user->notify(new ApiEndpointCalledNotification(Route::currentRouteName()));
        } catch (\Throwable $t) {
            return new JsonResource([
                $t->getMessage()
            ]);
        }

        return new JsonResource([
            'Notification send'
        ]);
    }
}
```

Testing Notifications

```php
public function testNotificationWasSend(): void
{
    Notification::fake();
    $user = User::factory()->create();

    $routeName = 'make-notification';
    $this->getJson(
        $this->urlGenerator->route($routeName)
    )->assertOk();

    Notification::assertSentTo(
        [$user],
        notification: ApiEndpointCalledNotification::class,
        function ($notification, $channels, $notifiable) use ($user, $routeName) {
            // Check channels
            $this->assertContains( needle: MailChannel::class, $channels);

            // Check the data that is passed to the view
            $notificationArray = $notification->toArray($user);
            $this->assertEquals($routeName, $notificationArray['route']);

            // Check mail attributes
            $mailNotification = (object)$notification->toMail($user);
            $this->assertEquals( expected: ApiEndpointCalledNotification::SUBJECT, $mailNotification->subject);
            $this->assertEquals( expected: ApiEndpointCalledNotification::TEMPLATE, $mailNotification->markdown);
            $this->assertEquals($notificationArray, $mailNotification->viewData);

            return true;
        }
    );
}
```

Testing Notifications

```php
public function testNotificationWasQueued(): void
{
    Queue::fake();

    $user = User::factory()->create();

    $this->getJson(
        $this->urlGenerator->action( action: NotificationAction::class)
    )->assertOk();

    // assertNotPushed if notification doesn't implement Illuminate\Contracts\Queue\ShouldQueue
    Queue::assertPushed(
        job: SendQueuedNotifications::class,
        function ($job, $queue, $data) use ($user) {
            $this->assertEquals( expected: ApiEndpointCalledNotification::class, get_class($job->notification));
            $this->assertEquals( expected: QUEUE_NOTIFICATIONS, $queue);
            $this->assertContains($user->id, $job->notifiables->pluck('id'));

            return true;
        }
    );
}
}
```

Files what to check:

Download:

- File name
- File content

Upload:

- File validation(size/mime/dimentions/etc)

Testing Files(Upload)

```php
final class UploadFileAction extends Controller
{
    public function __invoke(UploadFileRequest $request): JsonResource
    {
        if (!$request->image) {
            // remove file from DB ...
            // Storage::delete($user->file)

            return new JsonResource([
                'removed'
            ]);
        }


        $path = Storage::putFile(
            path: \App\Infrastructure\Storage::KEY_STORAGE_PATH,
            $request->image
        );

        return new JsonResource([
            $path
        ]);
    }
}
```

Testing Files(Upload)

```php
/**
 * @property-read File|UploadedFile|string|null $image
 */
final class UploadFileRequest extends ApiRequest
{
    public const KEY_IMAGE = 'image';

    public const VALIDATION_MIN_WIDTH = 100;
    public const VALIDATION_MAX_WIDTH = 2000;

    public const VALIDATION_MIN_HEIGHT = 100;
    public const VALIDATION_MAX_HEIGHT = 2000;

    public const VALIDATION_MAX_SIZE = 2048;

    public const VALIDATION_ALLOWED_MIMES = ['jpg', 'png', 'jpeg', 'gif', 'svg'];

    public function rules(): array
    {
        return [
            self::KEY_IMAGE => [
                'sometimes',
                'nullable',
                'image',
                'mimes:' . implode( separator: ',', array: self::VALIDATION_ALLOWED_MIMES),
                'max:' . self::VALIDATION_MAX_SIZE,
                'dimensions:min_width=' . self::VALIDATION_MIN_WIDTH . ',min_height=' . self::VALIDATION_MIN_HEIGHT
                . ',max_width=' . self::VALIDATION_MAX_WIDTH . ',max_height=' . self::VALIDATION_MAX_HEIGHT,
            ]
        ];
    }
}
```

Testing Files(Upload)

```php
public function testUploadImageSuccess(): void
{
    $storage = Storage::fake();

    // GD library required
    $fakeImage = UploadedFile::fake()->image(
        name: 'photo1.jpg',
        $this->faker->numberBetween(
            int1: UploadFileRequest::VALIDATION_MIN_WIDTH,
            int2: UploadFileRequest::VALIDATION_MAX_WIDTH
        ),
        $this->faker->numberBetween(
            int1: UploadFileRequest::VALIDATION_MIN_HEIGHT,
            int2: UploadFileRequest::VALIDATION_MAX_HEIGHT
        ),
    );

    $response = $this->postJson($this->urlGenerator->route( name: 'upload-file'), [
        UploadFileRequest::KEY_IMAGE => $fakeImage
    ]);

    $response->assertOk();
    $storage->assertExists($response->json( key: 'data.0'));
}
```

Testing Files(Upload)

```php
/**
 * @dataProvider providerWrongDimensions
 */
final public function testUploadImageDimensionErrors(int $width, int $height): void
{
    Storage::fake();

    $fakeImage = UploadedFile::fake()->image(
        name: 'photo1.jpg',
        $width,
        $height,
    );

    $response = $this->postJson($this->urlGenerator->route( name: 'upload-file'), [
        UploadFileRequest::KEY_IMAGE => $fakeImage
    ]);

    $response->assertStatus( status: Response::HTTP_UNPROCESSABLE_ENTITY);
    $this->assertContains(
        __( key: 'validation.dimensions', ['attribute' => UploadFileRequest::KEY_IMAGE]),
        $response->json( key: 'errors.' . UploadFileRequest::KEY_IMAGE)
    );
}
```

```php
final public function providerWrongDimensions(): iterable
{
    yield 'less_than_min' => [
        UploadFileRequest::VALIDATION_MIN_WIDTH - 1,
        UploadFileRequest::VALIDATION_MIN_HEIGHT - 1
    ];

    yield 'greater_than_max' => [
        UploadFileRequest::VALIDATION_MAX_WIDTH + 1,
        UploadFileRequest::VALIDATION_MAX_HEIGHT + 1
    ];
}
```

Testing Files(Upload)

```php
final public function testUploadImageSizeErrors(): void
{
    Storage::fake();

    $fakeImage = UploadedFile::fake()->create(
        name: 'photo1.jpg',
        kilobytes: UploadFileRequest::VALIDATION_MAX_SIZE + 1
    );

    $response = $this->postJson($this->urlGenerator->route( name: 'upload-file'), [
        UploadFileRequest::KEY_IMAGE => $fakeImage
    ]);

    $response->assertStatus( status: Response::HTTP_UNPROCESSABLE_ENTITY);
    $this->assertContains(
        __( key: 'validation.max.file', [
            'attribute' => UploadFileRequest::KEY_IMAGE,
            'max' => UploadFileRequest::VALIDATION_MAX_SIZE
        ]),
        $response->json( key: 'errors.' . UploadFileRequest::KEY_IMAGE)
    );
}
```

Testing Files(Upload)

```php
final public function testUploadImageMimeErrors(): void
{
    Storage::fake();

    $fakeImage = UploadedFile::fake()->create(
        name: 'photo1.txt',
        $this->faker->numberBetween( int1: 1, int2: UploadFileRequest::VALIDATION_MAX_SIZE),
    );

    $response = $this->postJson($this->urlGenerator->route( name: 'upload-file'), [
        UploadFileRequest::KEY_IMAGE => $fakeImage
    ]);

    $response->assertStatus( status: Response::HTTP_UNPROCESSABLE_ENTITY);
    $this->assertContains(
        __( key: 'validation.mimes', [
            'attribute' => UploadFileRequest::KEY_IMAGE,
            'values' => implode( separator: ', ', array: UploadFileRequest::VALIDATION_ALLOWED_MIMES)
        ]),
        $response->json( key: 'errors.' . UploadFileRequest::KEY_IMAGE)
    );
}
```

Testing Files(Download)

```php
class DownloadFileAction
{
    public const FILENAME = 'test';

    public function __invoke(): StreamedResponse
    {
        return response()->streamDownload(function () {
            echo $this->generateFile();
        }, name: self::FILENAME . '.csv');
    }

    private function generateFile(): string
    {
        $maxMemoryMb = 1024 * 1024 * 3;
        $handle = fopen( filename: 'php://temp/maxmemory:' . $maxMemoryMb, mode: 'w');
        $fields = [
            'column1' => 'value1',
            'column2' => 'value2',
        ];

        // Header
        fputcsv($handle, array_keys($fields));

        // Body
        fputcsv($handle, $fields);

        rewind($handle);
        $csv = stream_get_contents($handle);

        fclose($handle);

        return $csv;
    }
}
```

Testing Files(Download)

```php
public function testDownloadSuccess(): void
{
    $this->getJson($this->urlGenerator->route( name: 'download-file'))
        ->assertOk()
        ->assertDownload( filename: DownloadFileAction::FILENAME . '.csv')
    ;
}
```

Testing Files(Download)

```php
public function testFileContent(): void
{
    $content = $this->getJson($this->urlGenerator->route( name: 'download-file'))->streamedContent();
    $content = explode( separator: PHP_EOL, $content);

    // last line is empty
    array_pop( &array: $content);

    $header = null;
    foreach ($content as $key => $line) {
        if ($key === 0){
            $header = $line;
        } else {
            $data = array_combine(str_getcsv($header), str_getcsv($line));
            $this->debug([
                '$data' => $data,
                '$header' => $header
            ]);

            // Your file content checking logic goes here:
            $this->assertEquals(
                 expected: 'value1',
                $data['column1']
            );

            $this->assertEquals(
                 expected: 'value2',
                $data['column2']
            );
        }
    }
}
```

Response structures what to check:

- Each field inside reponse

Testing response structures

```php
Route::middleware(['auth:sanctum'])->group(function () {
    /** @uses App\Http\Actions\ActionWithResourceResponse\ResourceAction */
    Route::get( uri: '/user/{with_group?}', action: 'App\Http\Actions\ActionWithResourceResponse\ResourceAction')
        ->name( name: 'user-info')
    ;
});
```

```php
final class ResourceAction extends Controller
{
    public function __invoke(Authenticatable $user, $withGroup = null): UserResource|UserWithGroupResource
    {
        assert( assertion: $user instanceof User);

        return (bool)$withGroup ? new UserWithGroupResource($user) : new UserResource($user);
    }
}
```

Testing response structures

```php
/** @mixin \App\Models\User */
final class UserWithGroupResource extends JsonResource
{
    public const KEY_FIRSTNAME = User::COLUMN_FIRSTNAME;
    public const KEY_LASTNAME = User::COLUMN_LASTNAME;
    public const KEY_USERNAME = User::COLUMN_USERNAME;
    public const KEY_EMAIL = User::COLUMN_EMAIL;
    public const KEY_IMAGE = User::COLUMN_IMAGE;
    public const KEY_GROUP = User::RELATION_GROUP;

    public function toArray($request): array
    {
        return [
            self::KEY_FIRSTNAME => $this->firstname,
            self::KEY_LASTNAME => $this->lastname,
            self::KEY_USERNAME => $this->username,
            self::KEY_EMAIL => $this->email,
            self::KEY_IMAGE => $this->image,
            self::KEY_GROUP => new GroupResource($this->group),
        ];
    }
}
```

Testing response structures

```php
/* @mixin \App\Models\Group */
final class GroupResource extends JsonResource
{
    public const KEY_ID = Group::COLUMN_ID;
    public const KEY_NAME = Group::COLUMN_NAME;
    public const KEY_CODE = Group::COLUMN_CODE;

    public function toArray($request): array
    {
        return [
            self::KEY_ID => $this->id,
            self::KEY_NAME => $this->name,
            self::KEY_CODE => $this->code,
        ];
    }
}
```

Testing response structures

```
7  abstract class Structure
8  {
9      abstract public function getStructure(): array;
10
11     final public function getCollectionStructure(): array
12     {
13         return ['*' => $this->getStructure()];
14     }
15 }
```

Testing response structures

```php
final class UserWithGroupStructure extends Structure
{
    public function getStructure(): array
    {
        return [
            UserResource::KEY_FIRSTNAME,
            UserResource::KEY_LASTNAME,
            UserResource::KEY_USERNAME,
            UserResource::KEY_EMAIL,
            UserResource::KEY_IMAGE,
            UserWithGroupResource::KEY_GROUP => (new GroupStructure())->getStructure()
        ];
    }
}
```

Testing response structures

```php
final class ResponseStructureTest extends TestCase
{
    use WithAuthentication;

    public function testResponseStructure(): void
    {
        $user = User::factory()->create();
        $this->authAsUser($user);

        $groupShouldBeInResponse = $this->faker->boolean();
        $response = $this->getJson($this->urlGenerator->route( name: 'user-info', [
            'with_group' => $groupShouldBeInResponse
        ]));
        $response->assertOk();

        $structure = $groupShouldBeInResponse
            ? new UserWithGroupStructure()
            : new UserStructure()
        ;

        $response->assertJsonStructure([
            /** @see Structure::getCollectionStructure() for collections **/
            JsonResource::$wrap => $structure->getStructure()
        ]);
    }
}
```

Interacting With Time

```php
final class TimeAction extends Controller
{
    public const INVASION_DATE = '2022-02-24';

    public const RESPONSE_DEFAULT = '❤️ ';
    public const RESPONSE_AFTER_INVASION = 'Russian 🇷🇺 warship go f**ck yourself!';

    public function __invoke(): JsonResource
    {
        $response = TimeAction::RESPONSE_DEFAULT;

        $currentTime = Carbon::now();
        $invasionDate = Carbon::createFromFormat( format: 'Y-m-d', time: TimeAction::INVASION_DATE);

        if ($currentTime->gt($invasionDate)) {
            $response = TimeAction::RESPONSE_AFTER_INVASION;
        }

        return new JsonResource([$response]);
    }
}
```

Interacting With Time

```php
final class TimeMachineTest extends TestCase
{
    public function testMessageBasedOnCurrentDate(): void
    {
        $currentDate = $this->faker->dateTimeBetween( startDate: '-2 years', endDate: '+2 years')->format( format: 'Y-m-d');

        $now = Carbon::createFromFormat( format: 'Y-m-d', $currentDate)
            ->setTime( hour: 0, minute: 0)
        ;
        $this->travelTo($now);

        $expectedResponse = ($currentDate < TimeAction::INVASION_DATE)
            ? TimeAction::RESPONSE_DEFAULT
            : TimeAction::RESPONSE_AFTER_INVASION
        ;

        $url = $this->urlGenerator->action( action: TimeAction::class);

        $this->getJson($url)
            ->assertOk()
            ->assertExactJson([
                JsonResource::$wrap => [
                    $expectedResponse
                ],
            ])
        ;

        $this->debug([$currentDate, $expectedResponse]);
    }
}
```

Parallel running

```
> php artisan test --parallel
```

Laravel automatically handles creating and migrating a test database for each parallel process that is running your tests.

The test databases will be suffixed with a process token which is unique per process.

Code coverage

```
> XDEBUG_MODE=coverage php artisan test --parallel --coverage
```

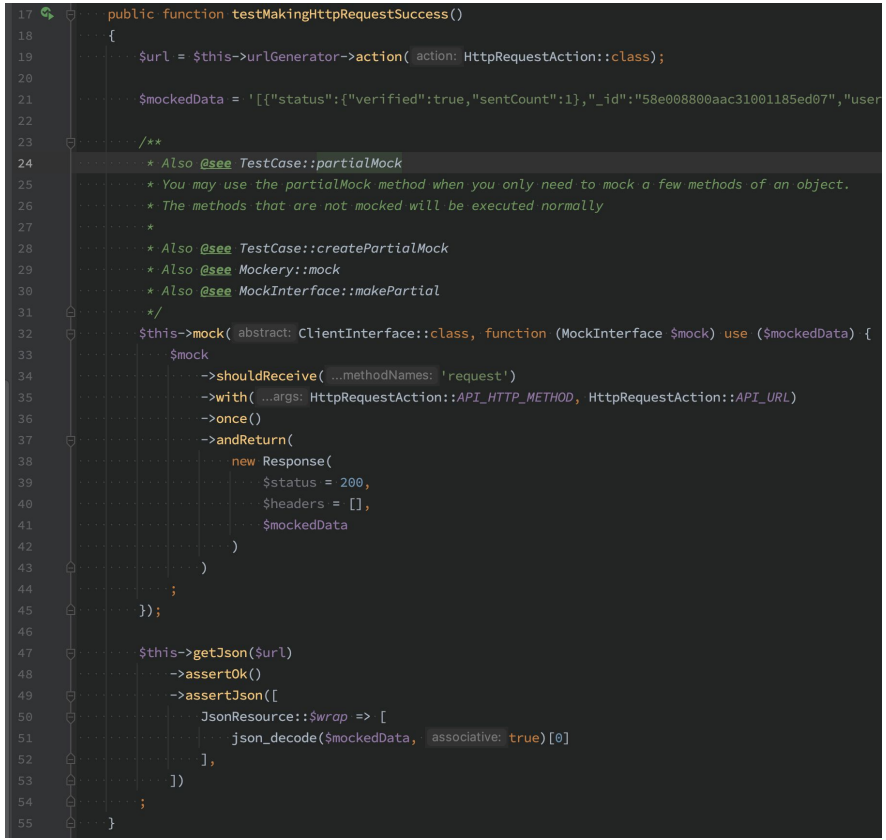
phpunit.xml

```
15      <coverage processUncoveredFiles="true">
16          <include>
17              <directory suffix=".php">./app</directory>
18          </include>
19          <report>
20              <html outputDirectory="tests/Coverage/html"/>
21          </report>
22      </coverage>
```

Mocking behavior

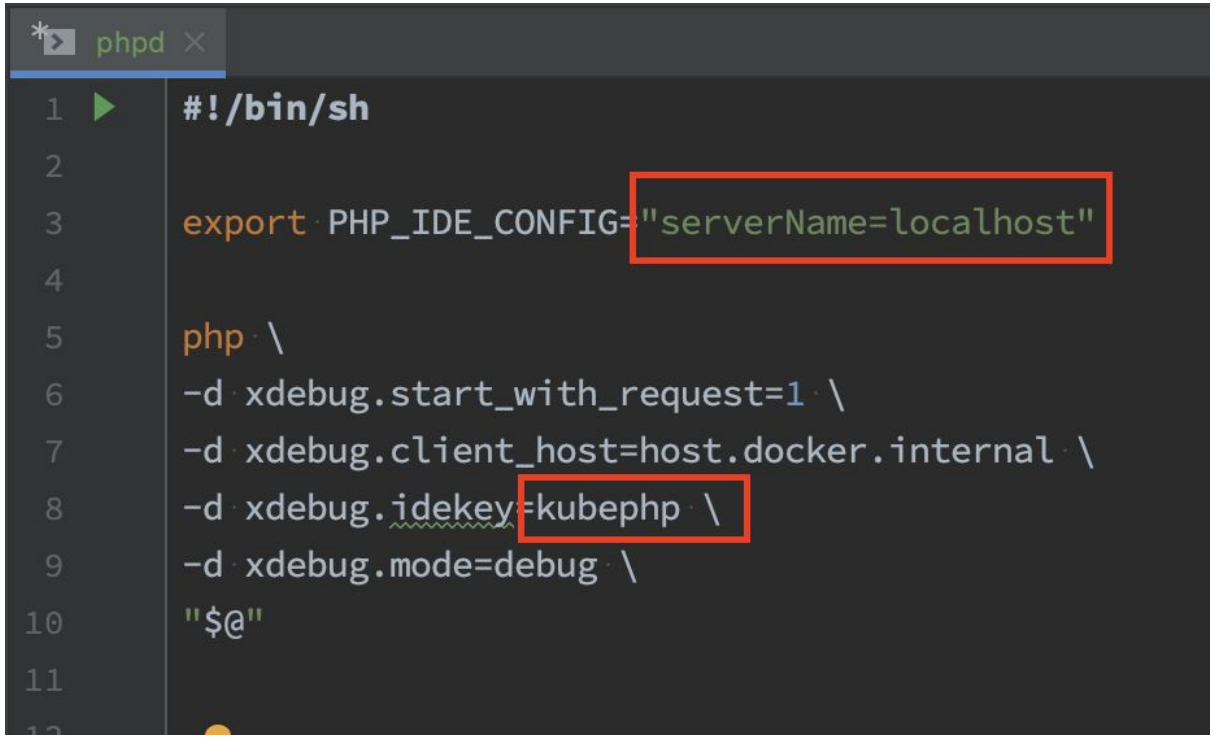
```php
final class HttpRequestAction extends Controller
{
    public const API_URL = 'https://cat-fact.herokuapp.com/facts';
    public const API_HTTP_METHOD = 'GET';

    public function __construct(
        private ClientInterface $client,
        private LoggerInterface $logger
    ) {
    }

    public function __invoke(): JsonResource
    {
        try {
            $facts = $this->client->request(HttpRequestAction::API_HTTP_METHOD, HttpRequestAction::API_URL);
        } catch (GuzzleException $e) {
            $this->logger->error('Request failed', ['error' => $e]);

            throw $e;
        }

        return new JsonResource(json_decode($facts->getBody()->getContents()));
    }
}
```

Mocking behavior

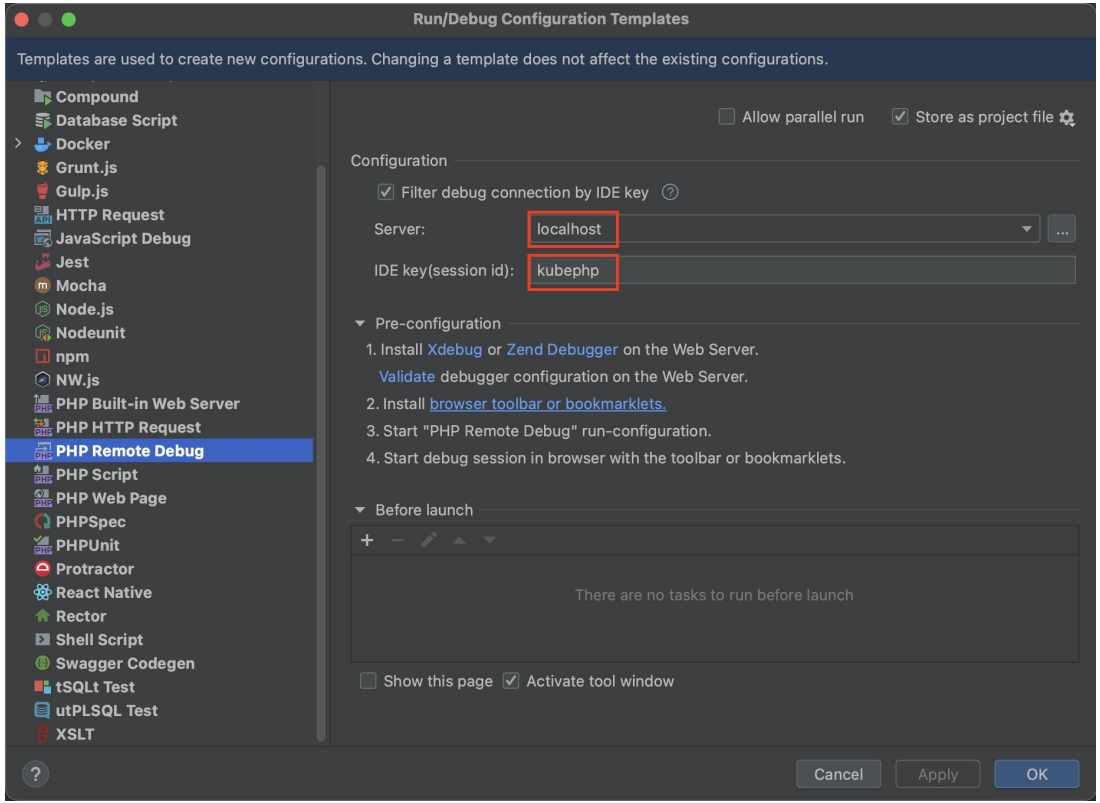


```php
public function testMakingHttpRequestSuccess()
{
    $url = $this->urlGenerator->action( action: HttpRequestAction::class);

    $mockedData = '[{"status":{"verified":true,"sentCount":1},"_id":"58e008800aac31001185ed07","user

    /**
     * Also @see TestCase::partialMock
     * You may use the partialMock method when you only need to mock a few methods of an object.
     * The methods that are not mocked will be executed normally
     *
     * Also @see TestCase::createPartialMock
     * Also @see Mockery::mock
     * Also @see MockInterface::makePartial
     */
    $this->mock( abstract: ClientInterface::class, function (MockInterface $mock) use ($mockedData) {
        $mock
            ->shouldReceive( ...methodNames: 'request')
            ->with( ...args: HttpRequestAction::API_HTTP_METHOD, HttpRequestAction::API_URL)
            ->once()
            ->andReturn(
                new Response(
                    $status = 200,
                    $headers = [],
                    $mockedData
                )
            )
        ;
    });

    $this->getJson($url)
        ->assertOk()
        ->assertJson([
            JsonResource::$wrap => [
                json_decode($mockedData, associative: true)[0]
            ],
        ])
    ;
}
```

Bonus: **Debugging CLI**



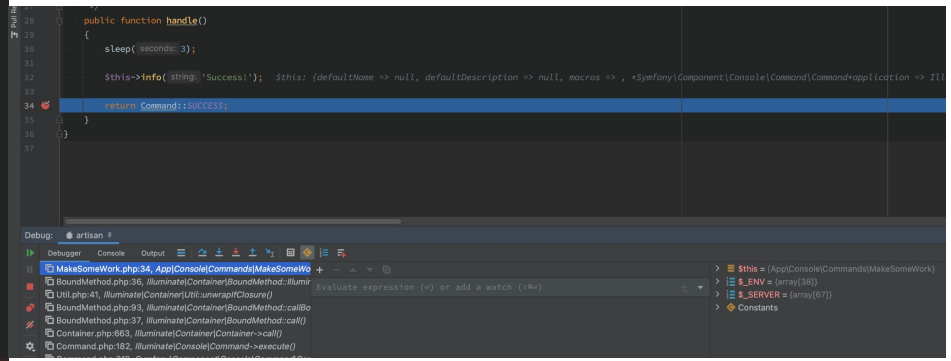
Bonus: **Debugging CLI**



Bonus: **Debugging CLI**

```php
class MakeSomeWork extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'make:work';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Command description';

    /**
     * Execute the console command.
     *
     * @return int
     */
    public function handle()
    {
        sleep( seconds: 3);

        $this->info( string: 'Success!');

        return Command::SUCCESS;
    }
}
```

```
/app $ php artisan make:work
Success!
/app $ ./phpd artisan make:work
```



Memory issues with GitHub Actions

- If you will face memory issues with GitHub Actions during running tests just split execution into multiple steps:

```
> php artisan test --env=testing tests/Feature/Admin
> php artisan test --env=testing tests/Feature/Auth
…
```

Tips:

- Don't use the Laravel App when you don't need to: If your tests extend from **Tests\TestCase** which uses the **CreatesApplication** trait, a new Laravel App is built for each test which takes a bit of time. This includes initialising the service-providers, detecting routes, and more.

  If you don't need Laravel's functionality in a test, extend from **PHPUnit\Framework\TestCase** instead.

Useful links:

- https://laravel.com/docs/9.x/testing
- https://blog.martinhujer.cz/how-to-use-data-providers-in-phpunit/
- https://www.code-distortion.net/books/fast-test-databases/
- https://mayahi.net/laravel/make-refresh-database-trait-much-faster/
- https://prinsfrank.nl/2022/09/20/How-to-write-decoupled-unit-tests-in-Laravel
- https://laravelpackage.com/
- https://devinthewild.com/blog/2022-05-31/testing-that-event-listeners-are-dispatched-in-laravel
- https://dev.to/macelux/how-to-test-mail-and-database-notifications-in-laravel-8-178o
- https://ralphjsmit.com/test-laravel-notification-e-mail-contents

Thank you !

Authored By Liamin Serhii