Cahier de bord

SimpleRouter.php

Samantha

SimpleRouter.php

J'ai fait le constructeur et ajouté des commentaires.

```php
public function __construct(Renderer $engine)
{
    // On vérifie que l'objet passé ($engine) est bien une instance de la classe Renderer
    $reflect = new \ReflectionClass($engine);
    $engineClass = $reflect->getName();

    // Si l'objet n'est pas une sous-classe de Renderer, on lance une exception pour signaler l'erreur
    if (!$reflect->isSubclassOf(Renderer::class)) {
        throw new \InvalidArgumentException("L'objet passé doit être une instance de Renderer, mais c'est une instance de $engineClass.");
    }

    // Si tout est correct, on enregistre l'objet Renderer dans une propriété pour l'utiliser dans la classe
    $this->engine = $engine;
}
```

Ensuite pour faire des Tests sur mon constructeur j'ai dû ajouter deux fichiers :

TestRenderer.php dans Template

SimpleRouterTest.php dans tests.

Explication de l'utilité de chaque fichier :

- *TestRenderer.php*

***TestRenderer* sert à simuler le comportement de la classe *Renderer* pendant les tests.** Même si c'est un fichier de test, je l'ai mis dans le dossier Template parce qu'il est lié à la gestion du rendu. Cela nous permet de tester d'autres parties du code, comme la classe *SimpleRouter*, sans avoir à gérer un vrai moteur de rendu.

- *SimpleRouterTest.php*

SimpleRouterTest permet de s'assurer que la classe *SimpleRouter* fonctionne bien.

Quoi faire avec ces fichiers ?

/! Pour tester vos fonctions, il faut obligatoirement se servir de TestSimpleRouter. **/!**

Pour TestRenderer, vous n'avez pas à y rajouter des fonctions.

Faire les tests :

- Créer une fonction de test dans SimpleRouterTest.php comme constructeur
- Ouvrir le terminal et se rendre dans le dossier du projet cloné avec **cd.**
- Installer PHPUnit avec Composer via la commande :

```
composer require --dev phpunit/phpunit ^9.5
```

- Exécuter les tests avec la commande :
  `php vendor/bin/phpunit --bootstrap vendor/autoload.php tests/SimpleRouterTest.php`
- Vérifier les résultats des tests dans le terminal.

Hudayfa

SimpleRouter.php

J'ai créé la méthode call et j'ai aussi créé une fonction dans SimpleRouterTest.php pour la tester.

Cette méthode `call` vérifie d'abord que la classe de la vue spécifiée existe et hérite correctement de `BaseView`. Elle crée ensuite une instance de cette vue et appelle sa méthode `render` avec la requête passée en paramètre pour récupérer le contenu. Si le contenu retourné est déjà une instance de `Response`, il est directement renvoyé. Sinon, une nouvelle instance de `Response` est créée avec ce contenu et renvoyée.

```php
    public function call(Request $request, ?Renderer $engine): Response
{
    // Vérifie si la classe existe et si elle est une sous-classe de BaseView
    $reflect = new \ReflectionClass($this->view);

    if (!$reflect->isSubclassOf(self::VIEW_CLASS)) {
        throw new \RuntimeException("La classe {$this->view} doit hériter de " . self::VIEW_CLASS);
    }

    // Crée une instance de la vue
    $viewInstance = new $this->view();

    // Vérifie que la méthode render existe
    if (!method_exists($viewInstance, self::VIEW_RENDER_FUNC)) {
        throw new \RuntimeException("La vue {$this->view} doit implémenter la méthode " . self::VIEW_RENDER_FUNC);
    }

    // Appelle la méthode render et récupère le contenu
    $content = $viewInstance->{self::VIEW_RENDER_FUNC}($request);

    // Si le contenu est déjà une instance de Response, on la retourne directement
    if ($content instanceof Response) {
        return $content;
    }

    // Crée et retourne la réponse HTTP avec le contenu
    return new Response($content);
}
```

Julien

SimpleRouter.php - function register

J'ai fait le register et j'ai ajouté des commentaires.

Cette fonction permet d'enregistrer une nouvelle route dans le routeur. Pour cela, elle prend en paramètre le chemin $path ainsi qu'un chemin ou une vue ($class_or_view).

Pour cela, elle vérifie si la route a déjà été rentré dans le tableau des routes ($this->routes). Si ce n'est pas le cas une nouvelle instance de route est créer.

```php
// Enregistre une route avec une vue ou un contrôleur.
public function register(string $path, string|object $class_or_view)
{
    // Permet de verifier si une meme route a deja été donné
    if (isset($this->routes[$path])) {
        throw new \InvalidArgumentException("Une route existe pour : $path.");
    }
    // Permet l'enregistrement de la route
    $this->routes[$path] = new Route($class_or_view);
}
```

Function Serve

```php
// Exécute l'action associée à une route
public function serve(mixed ...$args): void
{

// Obtenir la requête HTTP depuis Symfony
$request = Request::createFromGlobals();
$path = $request->getPathInfo();  // Récupèration du chemin de l'URL

// Verification de la route
if (!isset($this->routes[$path])) {
    throw new RouterException\RouteNotFoundException("La route '$path' n'a pas été trouvé.");
}

// Obtenir la route et exécuter la méthode call()
$route = $this->routes[$path];
$response = $route->call($request, $this->engine);

// Envoyer la réponse HTTP
$response->send();
}
```

Cette fonction `serve` permet de traiter une requête HTTP et d'envoyer une réponse.

Elle récupère d'abord l'URL demandée (le chemin `$path`) et vérifie si une route correspondante a été enregistrée dans le tableau des routes (`$this->routes`). Si aucune route n'est trouvée, elle déclenche une erreur.

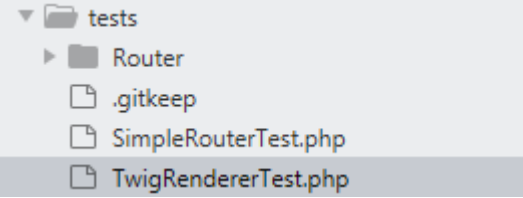
Si la route existe, elle exécute l'action associée (par exemple, une vue ou un contrôleur) via la méthode `call()` de l'objet `Route`. Enfin, elle renvoie la réponse générée au client.

TwigRenderer.php

Samantha

TwigRenderer.php

J'ai d'abord ajouté un fichier *TwigRendererTest.php* pour pouvoir tester les fonctions.



Et créer une fonction de test pour tester le constructeur.

J'ai fait le **constructeur** et ajouté des commentaires.

```php
//Initialise Twig avec le chemin des fichiers templates
public function __construct(string $templatePath)
{
    // Charger les fichiers de template depuis $templatePath
    $loader = new FilesystemLoader($templatePath);

    // Initialiser Twig avec ce chargeur
    $this->twig = new Environment($loader);
}
```

J'ai dû rajouter une fonction *getTwig()* car la propriété *$twig* de la classe *TwigRenderer* est privée. Cela signifie qu'elle n'est pas accessible directement depuis l'extérieur de la classe, y compris dans les tests. En ajoutant *getTwig()*, j'ai pu accéder à l'instance de Twig\Environment dans mes tests pour vérifier qu'elle a bien été initialisée dans le constructeur.

Les tests ont fonctionné.

```
PHPUnit 9.5.0 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 00:00.066, Memory: 6.00 MB

OK (1 test, 1 assertion)
```

Faire les tests :

- Créer une fonction de test dans TwigRendererTest.php comme constructeur
- Ouvrir le terminal et se rendre dans le dossier du projet cloné avec **cd.**
- Installer PHPUnit avec Composer via la commande (si ce n'est pas déjà fait) :

```
composer require --dev phpunit/phpunit ^9.5
```

- Exécuter les tests avec la commande :

```
php vendor/bin/phpunit --bootstrap vendor/autoload.php tests/TwigRendererTest.php
```

- Vérifier les résultats des tests dans le terminal.

J'ai fait la fonction **register** et ajouté des commentaires.

```php
//Enregistre un alias pour un dossier de templates
public function register(string $tag): void
{
    try {
        //Ajoute un dossier à Twig pour qu'il puisse trouver les templates
        //On utilise le tag comme nom du dossier (alias)
        $this->twig->getLoader()->addPath('./templates/' . $tag, $tag);
    } catch (\Exception $e) {
        // Si ça échoue, on capte l'erreur et on affiche un message
        throw new \RuntimeException("Impossible d'enregistrer le chemin ou l'alias : " . $e->getMessage());
    }
}
```

Register permet d'ajouter un dossier à Twig en lui donnant un nom (alias). Cela permet ensuite d'accéder facilement aux fichiers de ce dossier en utilisant l'alias au lieu de spécifier tout le chemin.
Par exemple, si on enregistre le dossier admin comme alias, au lieu d'utiliser à chaque fois le chemin complet **./templates/admin/template.twig** pour accéder à un fichier situé dans ce dossier, on pourra simplement utiliser **admin/template.twig**

Ensuite, j'ai fait des tests dans TwigRendererTest.php et ils ont fonctionnés.

Explications TwigRenderer.php

Qu'est-ce que TwigRenderer ?

C'est une classe qui permet d'utiliser **Twig**, un moteur de templates.
Elle rend des **templates** (fichiers .twig) en y insérant des données dynamiques (par exemple, un titre ou un contenu).

Pourquoi utiliser TwigRenderer ?

Séparation de l'affichage et des données :

Au lieu d'écrire du HTML et des données dans un seul fichier, vous placez le **HTML** dans des templates, et vous y insérez les **données**.
Twig se charge de **combiner** les deux pour générer la page web.

Comment ça fonctionne ?

Exemple : Vous avez un template index.twig avec des variables comme {{ title }} et {{ content }}.

TwigRenderer remplace ces variables par des valeurs réelles, comme un titre ou un texte provenant de votre application.

Exemple avec du code :

Fichier templates/index.twig :

```
<!DOCTYPE html>
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
        <h1>{{ content }}</h1>
    </body>
</html>
```

Ensuite, dans le fichier PHP où vous souhaitez afficher le contenu, vous allez utiliser `TwigRenderer` pour rendre ce template avec les données.

Fichier index.php :

```php
<?php

// Assurez-vous d'inclure l'autoloader de Composer pour charger Twig et d'autres classes d
require_once 'vendor/autoload.php';

use Framework312\Template\TwigRenderer;

// 1. Créez une instance de TwigRenderer avec le chemin vers les templates
$renderer = new TwigRenderer('./templates');  // Le chemin où se trouvent les templates Tw

// 2. Les données à injecter dans le template
$data = [
    'title' => 'Ma page dynamique',  // Le titre de la page
    'content' => 'Bienvenue sur mon site!'  // Le contenu de la page
];

// 3. Rendre le template 'index.twig' avec les données
// Cela génère et affiche le HTML final
echo $renderer->render($data, 'index.twig');

?>
```

Résultat généré (HTML final) :

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Ma page dynamique</title>
    </head>
    <body>
        <h1>Bienvenue sur mon site!</h1>
    </body>
</html>
```

- **{{ title }}** sera remplacé par **"Ma page dynamique"**.
- **{{ content }}** sera remplacé par **"Bienvenue sur mon site!"**.

Pour générer une page HTML avec TwigRenderer, il y a besoin de deux fichiers :

- 1 fichier de **template** `.twig`.
- 1 fichier **PHP** qui utilise **`TwigRenderer`** pour rendre ce template avec les données.

Lorsqu'il faudra essayer que le html se soit bien généré, il faudra ouvrir dans le navigateur le index.php (dans cet exemple).

Hudayfa

TwigRenderer.php

J'ai créé la méthode render qui utilise Twig pour charger un template et le rendre avec les données fournies.

```php
    public function render(mixed $data, string $template): string
{
    try {
        // Utilisation de Twig pour charger et rendre le template avec les données
        return $this->twig->render($template, (array) $data);
    } catch (\Twig\Error\LoaderError $e) {
        throw new \RuntimeException("Erreur de chargement du template : {$e->getMessage()}", 0, $e);
    } catch (\Twig\Error\RuntimeError $e) {
        throw new \RuntimeException("Erreur à l'exécution du template : {$e->getMessage()}", 0, $e);
    } catch (\Twig\Error\SyntaxError $e) {
        throw new \RuntimeException("Erreur de syntaxe dans le template : {$e->getMessage()}", 0, $e);
    }
}
```

Elle commence par appeler la méthode `render` de Twig, en convertissant les données en tableau si nécessaire. Si une erreur survient (comme un problème de chargement, d'exécution, ou de syntaxe dans le template), elle est interceptée et encapsulée dans une exception `RuntimeException` avec un message détaillant l'erreur spécifique.
Ensuite je l'ai testé dans le fichier TwigRendererTest.php

```php
    public function testRenderWithValidData()
{
    // Crée un dossier temporaire pour les templates
    $templatePath = __DIR__ . '/templates';
    if (!is_dir($templatePath)) {
        mkdir($templatePath);
    }

    // Crée un fichier template de test
    $templateFile = $templatePath . '/test.twig';
    file_put_contents($templateFile, '<h1>{{ title }}</h1><p>{{ content }}</p>');

    // Initialise TwigRenderer avec le chemin
    $renderer = new TwigRenderer($templatePath);

    // Données à injecter dans le template
    $data = ['title' => 'Bienvenue', 'content' => 'Bonjour, monde !'];

    // Appelle la méthode render
    $result = $renderer->render($data, 'test.twig');

    // Vérifie le contenu retourné
    $this->assertStringContainsString('<h1>Bienvenue</h1>', $result);
    $this->assertStringContainsString('<p>Bonjour, monde !</p>', $result);

    // Nettoyage après le test
    unlink($templateFile);
    rmdir($templatePath);
}
```

Le test vérifie le bon fonctionnement de la méthode render de TwigRenderer. Il commence par créer un dossier temporaire pour les templates, y ajoute un fichier test.twig contenant un exemple de HTML avec des variables Twig. Ensuite, il initialise un objet TwigRenderer avec le chemin du dossier et injecte des données dans le template via la méthode render. Le test compare le résultat produit avec le HTML attendu, notamment en vérifiant que les balises <h1> et <p> contiennent les valeurs injectées. À la fin, le fichier template et le dossier sont supprimés pour nettoyer l'environnement.

Les fichiers de vue

Hugo

View/HTMLView.php

La vue la plus simple du micro-framework, car elle ne nécessite aucune transformation des données retournées.

```php
2 usages  1 inheritor  Hugo Lignères
class HTMLView extends BaseView
{
    // on indique si cette vue utilise ou non un moteur de template
    no usages  Hugo Lignères
    static public function use_template(): bool
    {
        // 'return true' car cette vuE utilise un moteur de template (Twig)
        return true;
    }

    // on génère et on renvoie une réponse HTTP avec un contenu HTML
    Hugo Lignères
    public function render(Request $request): Response
    {
        // On récupère la méthode HTTP utilisée dans la requête générée
        $method = strtolower($request->getMethod());

        // On vérifie que la méthode utilisée existe bien. Si c'est le cas, on pas à la suite
        if (!method_exists($this, $method)) {
            // Sinon, cela signifie que la méthode n'est pas implémentée ---> on lève une exception
            throw new \BadMethodCallException( message: "La méthode HTTP [$method] n'existe pas.");
        }

        // On appelle la méthode correspondante à la requête et on récupère son contenu
        $data = $this->$method($request);

        // On retourne une réponse HTTP avec du contenu HTML et un type de contenu text/html
        return new Response($data,  status: Response::HTTP_OK, ['Content-Type' => 'text/html']);
    }
}
```

View/JSONView.php

Cette classe a le même fonctionnement que HTMLView, avec deux exceptions cependant :

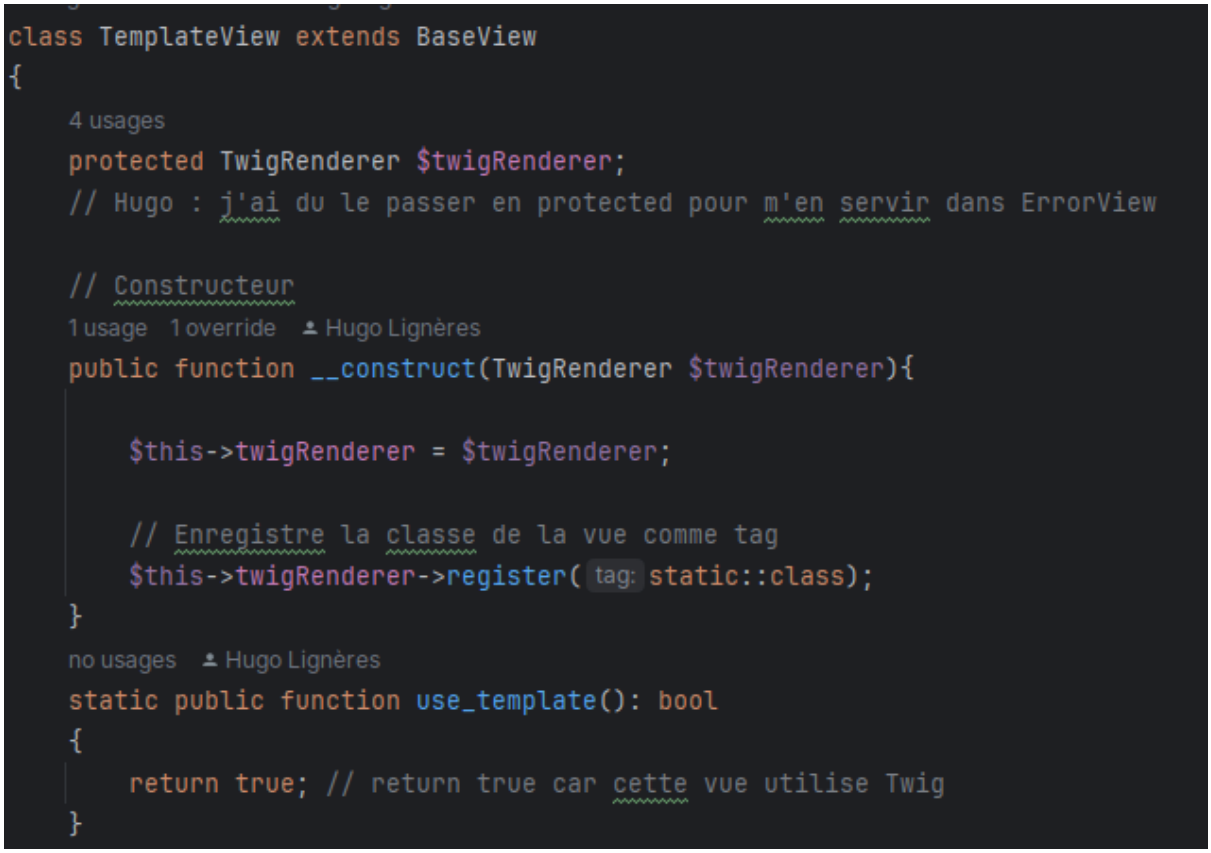
- Contrairement à HTMLView, cette vue n'utilise pas de template.

- La réponse avec retournée avec JsonResponse, car elle contient des méthodes qui permettent de bien traiter les données JSON retournées. Cela évite donc de rajouter du code en plus.

```php
class JSONView extends BaseView
{
    // on indique si cette vue utilise ou non un moteur de template (Twig)
    no usages  Hugo Lignères
    static public function use_template(): bool
    {
        // 'return false' car la vue n'a pas besoin d'utiliser Twig
        return false;
    }

    // on génère et on renvoie une réponse HTTP avec un contenu JSON
    Hugo Lignères
    public function render(Request $request): Response
    {
        // On récupère la méthode HTTP utilisée dans la requête générée
        $method = strtolower($request->getMethod());

        // On vérifie que la méthode utilisée existe bien. Si c'est le cas, on pas à la suite
        if(!method_exists($this, $method)) {
            // Sinon, cela signifie que la méthode n'est pas implémentée ---> on lève une exception
            throw new \BadMethodCallException( message: "La méthode HTTP [$method] n'existe pas.");
        }

        // On appelle la méthode correspondante à la requête et on récupère son contenu
        $data = $this->$method($request);

        // On retourne une réponse JSON avec les données dans ce format et un type 'application/json'
        // => ce type est déjà géré par 'JsonResponse', donc pas besoin de le préciser
        return new JsonResponse($data,  status: Response::HTTP_OK);
    }
}
```

View/TemplateView.php

La première partie de cette classe se passe dans le constructeur de cette dernière. On stocke une instance de TwigRenderer dans la variable $twigRenderer. On injecte la dépendance de twigRenderer, puis on l'enregistre dans la classe en tant que tag dans Twig.



```php
class TemplateView extends BaseView
{
    4 usages
    protected TwigRenderer $twigRenderer;
    // Hugo : j'ai du le passer en protected pour m'en servir dans ErrorView

    // Constructeur
    1 usage  1 override  Hugo Lignères
    public function __construct(TwigRenderer $twigRenderer){

        $this->twigRenderer = $twigRenderer;

        // Enregistre la classe de la vue comme tag
        $this->twigRenderer->register( tag: static::class);
    }
    no usages  Hugo Lignères
    static public function use_template(): bool
    {
        return true; // return true car cette vue utilise Twig
    }
```

La deuxième partie se trouve dans la fonction render(). Le fonctionnement de cette dernière est similaire aux deux classes précédentes, sauf à la fin. On doit créer une variable $template_name , qui définit dynamiquement le nom du fichier Twig, comme ceci : nom.html.twig. Seul le 'nom' change, car 'html.twig' correspond à l'extension des fichiers Twig. On retourne ensuite le contenu HTML en utilisant les différentes propriétés définies dans la classe TemplateView.

```php
    1 override  Hugo Lignères
    public function render(Request $request): Response
    {
        $method = strtolower($request->getMethod());

        // On vérifie que la méthode existe
        if(!method_exists($this, $method)){
            throw new \BadmethodCallException( message: "La méthode HTTP [$method] n'existe pas.");
        }

        // On appelle la méthode
        $data = $this->$method($request);

        // Exception si les données retournées par la méthode n'est pas un tableau
        if(!is_array($data)){
            throw new \RuntimeException( message: "Un tableau devrait être retourné par la méthode [$method]");
        }

        // Nom du template associé à l'extension du fichier Twig (ex: nom_template.html.twig)
        $template_name = str_replace( search: '\\', replace: '/', subject: static::class . '.html.twig');

        $html_content = $this->twigRenderer->render($data, $template_name);

        // Retourne une réponse avec le contenu HTML généré
        return new Response($html_content);
    }
}
```

View/ErrorView.php

Le rôle de ce fichier est de gérer les erreurs HTTP les plus fréquentes en créant une vue spécialement conçue pour cela. Cette vue génère du contenu qui est ensuite dirigé vers un fichier Twig, également conçu pour afficher une page HTML qui présente ces erreurs HTTP.

Il y avait deux approches possibles :

- Une approche dynamique, qui renvoi la même mise en forme de contenu, et où seul le message d'erreur change en fonction de l'erreur HTTP

- Une approche statique, avec une mise en forme unique pour chaque erreur HTTP

Nous avons choisi l'approche dynamique, car nous pensons qu'il n'y avait pas besoin de créer un contenu pour chaque erreur HTTP. Cela permet également de ne pas trop alourdir le framework.

Pour cela, on met en place un tableau associatif $errors[], qui contient un message attribué à chaque erreur HTTP.

```php
class ErrorView extends TemplateView
{
    // Messages d'erreurs - à modifier si besoin
    1 usage
    private array $errors = [
        401 => 'Erreur 401: Utilisateur non authentifié',
        403 => 'Erreur 403: Accès interdit',
        404 => 'Erreur 404: Page non trouvée',
        500 => 'Erreur 500: Erreur interne du serveur',
        502 => 'Erreur 502: Erreur de connexion',
        503 => 'Erreur 503: Service non disponible',
    ];
```

Ensuite, on se sert de deux variables : $statusCode qui contient le code de l'erreur, et $message qui contient le message associé à l'erreur HTTP. C'est dans le constructeur de la classe que l'on attribue ces données aux variables correspondantes. Ensuite, la fonction render() génère le contenu HTML contenant l'erreur et le message d'erreur associé, puis renvoie ce contenu HTML au fichier Twig conçu pour.

```php
    // Variables pour stocker l'erreur HTTP rencontrée et son message associé
    4 usages
    private int $statusCode;
    2 usages
    private string $message;

    // Constructeur qui associe ces variables aux données générées par l'erreur HTTP
    6 usages  Hugo Lignères
    public function __construct(TwigRenderer $twigRenderer, int $statusCode, ?string $message){

        parent::__construct($twigRenderer);

        $this->statusCode = $statusCode;
        $this->message = $message ?? $this->errors[$this->statusCode] ?? 'Erreur inconnue';
    }

    // Fonctionne qui génère le contenu HTML pour ensuite l'envoyer vers le fichier Twig
     Hugo Lignères
    public function render(Request $request): Response
    {

        $html_content = $this->twigRenderer->render(
            [
                'code' => $this->statusCode,
                'message' => $this->message,
            ],
            template: 'errors/error.html.twig'
        );

        // Réponse générée avec le contenu HTML
        return new Response($html_content, $this->statusCode);
    }
}
```

Fichiers des tests pour les vues

Pour s'assurer que les fichiers de vue fonctionnent correctement, la deuxième étape était de tester ces fichiers avec PHPUnit.

Tests/Router/View/HTMLViewTest.php , JSONViewTest.php, TemplateViewTest.php

Les trois premiers fichiers de test ont globalement la même structure :

1. On crée un mockTest en préciser la structure HTML qui doit être retournée

2. On crée une vue test, basée sur $twigRenderer et sur la classe qui est testée. Cette vue test précise, dans un tableau associatif, le statut et le message que la requête doit retourner.

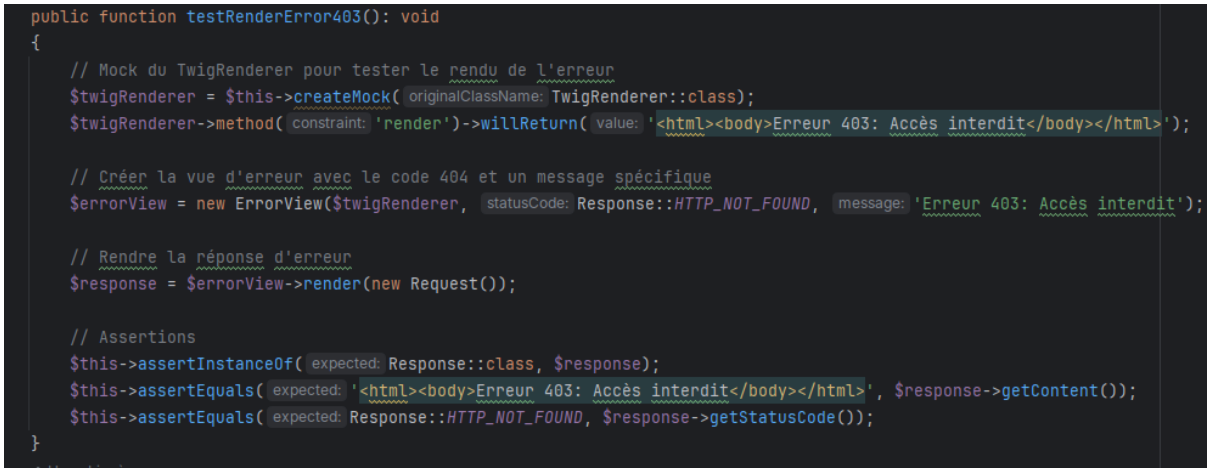
3. Enfin, on met en place des assertions, qui sont les éléments de ces fichiers que l'on souhaite tester.

Un exemple avec HTMLViewTest.php

```php
class HTMLViewTest extends TestCase {
    Hugo Lignères
    public function testRenderReturnHtmlResponse(): void
    {
        // On crée une instance de requête
        $request = new Request();

        // On crée une vue HTMLView, avec une méthode get pour le test
        $view = new class() extends HTMLView {
            Hugo Lignères
            protected function get(Request $request): string
            {
                // Contenu de la vue crée pour le test
                return '<html><body>Ça fonctionne!</body></html>';
            }
        };

        // On appelle la méthode "render"
        $response = $view->render($request);

        /* Assertions */

        // On vérifie que la réponse est une instance de Response
        $this->assertInstanceOf( expected: Response::class, $response);

        // On vérifie le type de la réponse, qui doit être text/html ici
        $this->assertEquals( expected: 'text/html', $response->headers->get( key: 'Content-Type'));

        // On vérifie que le contenu de la réponse est valide par rapport au format attendu
        $this->assertEquals( expected: '<html><body>Ça fonctionne!</body></html>', $response->getContent());

        // On vérifie que le statut de la requête HTTP est bien 200
        $this->assertEquals( expected: Response::HTTP_OK, $response->getStatusCode());
    }
}
```

Tests/Router/View/ErrorViewTest.php

Ce fichier possède une structure de tests similaire aux précédents fichiers de test, sauf que l'on applique cette structure à chacune des erreurs. Cela permet d'obtenir la plus grande précision possible pour chaque erreur HTTP. Voici un exemple pour l'erreur 403 :



```php
public function testRenderError403(): void
{
    // Mock du TwigRenderer pour tester le rendu de l'erreur
    $twigRenderer = $this->createMock( originalClassName: TwigRenderer::class);
    $twigRenderer->method( constraint: 'render')->willReturn( value: '<html><body>Erreur 403: Accès interdit</body></html>');

    // Créer la vue d'erreur avec le code 404 et un message spécifique
    $errorView = new ErrorView($twigRenderer,  statusCode: Response::HTTP_NOT_FOUND,  message: 'Erreur 403: Accès interdit');

    // Rendre la réponse d'erreur
    $response = $errorView->render(new Request());

    // Assertions
    $this->assertInstanceOf( expected: Response::class, $response);
    $this->assertEquals( expected: '<html><body>Erreur 403: Accès interdit</body></html>', $response->getContent());
    $this->assertEquals( expected: Response::HTTP_NOT_FOUND, $response->getStatusCode());
}
```

Fichiers de templates Twig

Templates/base.html.twig

C’est le fichier qu’il est primordial d’avoir dans le dossier templates. C'est lui qui est responsable de la structure HTML de la page. Il faut donc l'intégrer aux autres fichiers Twig pour retrouver cette structure. Cela permet de ne pas avoir à réécrire cette structure HTML dans chaque fichier Twig.

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ title }}</title>
</head>

<body>
<header>
    <h1>Framework UE312 - Groupe Level</h1>
</header>
<main>
    {% block content %}{% endblock %}
</main>

<footer>
    <p>Micro-framework conçu pour le projet de groupe de l'UE 312</p>
</footer>
</body>
</html>
```

Templates/index.html.twig

Ce fichier est un exemple de comment créer des fichiers Twig, autres que base.html.twig. Comme le montre la capture d'écran, il faut d'abord étendre le fichier base pour appliquer la structure HTML de la page au fichier index. Ensuite, le fichier index contient le contenu HTML qui sera généré uniquement par ce fichier.

```twig
{% extends "base.html.twig" %}

{% block title %}
    Bienvenue
{% endblock %}

{% block content %}
    <section>
        <p>Ceci est un exemple de page</p>
    </section>
{% endblock %}
```

Tests/errors/error.html.twig

C'est le fichier qui affiche le contenu généré par ErrorView.php

On y retrouve l'affichage du code avec la variable {{ code }} et l'affichage du message d'erreur avec {{ message }}. Ces variables proviennent du tableau associatif $html_content de ErrorView.php.

```twig
{% extends "base.html.twig" %}

{% block title %}

{% endblock %}
    Erreur: {{ code }}
{% endblock %}

{% block content %}
    <section>
        <h2>Erreur : {{ code }}</h2>
        <p>{{ message }}</p>
    </section>
{% endblock %}
```