

Rapport Maya Api

“Tests unitaires et d’intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

Sommaire :

- [Mise en place des tests unitaires](#)
- [Mise en place des tests d’intégrations](#)
- [Mise en place des tests fonctionnels](#)
- [Mise en place des tests de robustesse](#)
- [Mise en place des tests de performances](#)
- [Mise en place des tests de charges](#)
- [Mise en place des tests de sécurité](#)
- [Mise en place de la couverture des tests \(Code coverage\)](#)

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

1. MISE EN PLACE DES TESTS UNITAIRES

La première phase de la mission consistait à vérifier le fonctionnement isolé de nos classes PHP (Entités), sans interaction avec la base de données ou le framework Symfony complet.

1.1 Installation de l'infrastructure de test

Pour débiter, j'ai dû installer les outils nécessaires au lancement des tests :

- Exécution de la commande `composer require --dev phpunit/phpunit symfony/test-pack`. Cela installe **PHPUnit**, le moteur de test de référence en PHP.
- Vérification de la présence du fichier `phpunit.xml.dist` à la racine, qui définit les répertoires où se trouvent les tests (dossier `tests/`).

1.2 Création et configuration de `CategorieUnitTest`

J'ai créé le fichier `tests/Unit/Entity/CategorieUnitTest.php` en respectant les conventions de nommage.

- **Implémentation des tests d'accessors** : J'ai écrit des méthodes pour tester chaque `Getter` et `Setter` de l'entité `Categorie`. Par exemple, vérifier que la méthode `setLibelle()` assigne correctement la valeur et que `getLibelle()` la retourne.
- **Utilisation de données personnalisées** : Dans chaque test, j'ai utilisé mon prénom (ex: "Categorie de [Prénom]") pour garantir que les tests sont bien spécifiques à mon travail.
- **Test de logique métier** : J'ai testé la méthode `getImageUrl()` pour m'assurer qu'elle retourne bien une URL formatée ou `null` si aucune image n'est présente.

1.3 Développement de `ProduitUnitTest`

Sur le même modèle, j'ai développé les tests pour l'entité `Produit` :

- Vérification des champs spécifiques : Libellé, Description, Prix, et Disponibilité.
- Test de la relation : S'assurer qu'un produit peut être lié à une catégorie.

1.4 Phase de test d'échec (Red/Green)

Pour valider la robustesse de mes tests, j'ai volontairement introduit des erreurs :

- Modification d'une assertion pour attendre "Erreur" au lieu de mon prénom.
- Exécution via `php bin/phpunit` et constatation de l'échec (affichage rouge).

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- Correction du code pour repasser tous les tests au succès (affichage vert).

2. MISE EN PLACE DES TESTS D'INTÉGRATION

Une fois les unités de code validées, je suis passé à la phase d'intégration pour tester le comportement des entités au sein du framework (validation, base de données, upload de fichiers).

2.1 Configuration de l'environnement de base de données

Les tests d'intégration nécessitent une base de données réelle mais isolée :

- **Création du fichier `.env.test`** : Configuration d'une base de données nommée `maya26_test`.
- **Initialisation** : Création de la base via `php bin/console doctrine:database:create --env=test` et mise à jour du schéma avec les migrations.

2.2 Automatisation des données avec Zenstruck Foundry

Plutôt que de créer des objets manuellement, j'ai utilisé le design pattern **Factory** :

- Installation via `composer require --dev zenstruck/foundry`.
- Génération de `CategorieFactory` et `ProduitFactory`. Cela permet de créer des données de test en une ligne de code :
`CategorieFactory::createOne(['libelle' => '[Prénom]'])`.

2.3 Développement de `CategorieIntegrationTest`

Cette classe hérite de `KernelTestCase`, ce qui permet de "booter" Symfony durant le test.

- **Test du Validator** : J'ai récupéré le service `validator` pour vérifier que les contraintes d'entité fonctionnent (ex: une erreur doit apparaître si le libellé est vide ou trop court).
- **Test d'unicité** : J'ai testé la contrainte `UniqueEntity`. J'ai tenté d'insérer deux catégories avec le même libellé pour vérifier que le système bloque bien la seconde insertion.
- **Test du cycle de vie des images** : Utilisation de `VichUploader` pour simuler l'envoi d'un fichier image et vérifier qu'il est bien stocké et que le champ `imageDateMaj` est mis à jour.

2.4 Développement de `ProduitIntegrationTest`

J'ai appliqué la même méthodologie pour les produits :

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- Vérification que les prix négatifs sont refusés par le validateur.
- Test de la persistance en base de données : créer un produit, le sauvegarder, puis le rechercher via l'EntityManager pour vérifier l'exactitude des données.

2.5 Analyse des résultats finaux

Après avoir corrigé les erreurs introduites pour la démonstration, j'ai lancé la suite complète : `php bin/phpunit tests/Integration` Tous les tests ont été validés avec succès, garantissant que les fonctionnalités critiques de l'API (validation, unicité, stockage) sont opérationnelles.

```
C:\wamp64\www\maya-api>php bin/phpunit tests/Integration/Entity/CategorieIntegrationTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:      PHP 8.2.18
Configuration: C:\wamp64\www\maya-api\phpunit.dist.xml

.....                                             9 / 9 (100%)

Time: 00:03.635, Memory: 34.00 MB

OK (9 tests, 24 assertions)

C:\wamp64\www\maya-api>
```

```
public function testCreateCategorieWithImage(): void
{
    $categorie = new Categorie();
    $categorie->setLibelle(libelle: 'Test Creation');

    // Création fichier temporaire
    $tmpFile = tempnam(directory: sys_get_temp_dir(), prefix: 'vich');
    file_put_contents(filename: $tmpFile, data: 'image content');

    $uploadedFile = new UploadedFile(
        path: $tmpFile,
        originalName: 'test.jpg',
        mimeType: 'image/jpeg',
        error: null,
        test: true
    );
};
```

3. MISE EN PLACE DES TESTS FONCTIONNELS

3.1 Résumé des concepts et outils ajoutés

Pour cette dernière phase, l'objectif était de passer du test de composants isolés au test de l'application dans son ensemble (bout-en-bout). Les tests fonctionnels vérifient que les points d'entrée de l'API (Endpoints) répondent correctement aux requêtes HTTP.

Nouveaux outils et concepts utilisés :

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- **ApiTestCase** : Une classe spécifique fournie par API Platform qui étend les capacités de Symfony pour simuler un client HTTP performant et tester les réponses JSON.
 - **DAMA Doctrine Test Bundle** : Cet outil a été ajouté pour optimiser les tests en base de données. Il utilise des transactions SQL pour annuler (rollback) toutes les modifications après chaque test, garantissant une base toujours propre sans avoir à la recréer.
 - **ValidationExceptionListener** : Mise en place d'un écouteur d'événements pour intercepter les erreurs de validation Symfony et les transformer en réponses JSON standardisées (format Hydra/JSON-LD) avec un code de retour **422 (Unprocessable Entity)**.
 - **Client HTTP** : Utilisation de `static::createClient()` pour envoyer de véritables requêtes **GET**, **POST**, **PUT**, ou **DELETE** vers l'API.
-

3.2 Résumé des tests fonctionnels de l'entité **Categorie**

Les tests réalisés sur l'entité **Categorie** visaient à valider le comportement des routes API. Voici les principaux scénarios testés :

- **Récupération de la liste (GET)** : Vérification que l'API retourne bien l'ensemble des catégories avec un code **200 OK** et que le format JSON est conforme au standard Hydra.
- **Création d'une catégorie (POST)** : * *Cas nominal* : Envoi d'un libellé valide (contenant votre prénom) et vérification de la création en base (code **201 Created**).
 - *Cas d'erreur* : Envoi de données invalides (libellé vide ou trop court) pour vérifier que le `ValidationExceptionListener` intercepte bien l'erreur et retourne un code **422** avec le message d'erreur approprié.
- **Modification (PUT/PATCH)** : Vérification de la mise à jour d'une catégorie existante et de la persistance des changements.
- **Suppression (DELETE)** : Test de la suppression d'une catégorie par son ID et vérification que la ressource n'est plus accessible (code **204 No Content**).

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

```
C:\wamp64\www\maya-api>php bin/phpunit tests/functional/Entity/CategorieFunctionalTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:        PHP 8.2.18
Configuration: C:\wamp64\www\maya-api\phpunit.dist.xml

....                                                    4 / 4 (100%)

Time: 00:12.092, Memory: 76.00 MB

OK (4 tests, 16 assertions)

Faker seed used: 421383
C:\wamp64\www\maya-api>
```

3.3 Synthèse technique du fonctionnement

Le flux de test fonctionnel suit ce schéma :

1. **Requête** : Le test envoie une requête HTTP (ex: `POST /api/categories`).
2. **Traitement** : Symfony traite la requête, passe par le validateur et les écouteurs de succès ou d'exception.
3. **Réponse** : L'API retourne un flux JSON.
4. **Assertion** : Le test vérifie trois points : le **Code Status** (201, 422, etc.), le **Content-Type** (application/ld+json) et la **structure du contenu** (présence du libellé avec votre prénom).
5. **Assertion** : Le test vérifie trois points : le Code Status (201, 422, etc.), le Content-Type (application/ld+json) et la structure du contenu (présence du libellé avec votre prénom).

4. MISE EN PLACE DES TESTS DE ROBUSTESSE

Cette phase visait à tester la résilience du système face à des conditions extrêmes ou des données inattendues pour éviter tout crash brutal.

4.1 Robustesse au niveau de l'Entité

- **Normalisation (Trim)** : Constat qu'un libellé composé uniquement d'espaces passait la validation. J'ai donc modifié l'entité pour ajouter l'option `normalizer: 'trim'` afin de rejeter ces cas.

+2

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- **Fuzzing** : Injection de 200 chaînes de caractères aléatoires et spéciales dans le champ libellé pour vérifier l'absence d'exception PHP et la stabilité du validateur.
+1
- **Fichiers invalides** : Test de résistance lors de l'upload de fichiers non-images ou de tailles extrêmes (>2 Mo).
+1

4.2 Robustesse au niveau de l'API

- **Isolation de l'environnement** : Configuration d'un dossier d'upload spécifique pour les tests (`var/test_uploads`) via la surcharge du fichier `vich_uploader.yaml` dans l'environnement de test afin de ne pas polluer les images de développement.
+1
 - **Idempotence de suppression** : Vérification que tenter de supprimer deux fois la même catégorie retourne bien une erreur 404 gérée proprement au second passage.
+2
 - **Cas limites HTTP** : Tests de requêtes avec des paramètres manquants (libellé null) ou des noms de fichiers extrêmement longs (600 caractères) pour s'assurer que l'API reste stable.
+1
-

5. MISE EN PLACE DES TESTS DE PERFORMANCE

L'objectif de cette mission était de mesurer la rapidité et l'efficacité globale du système sous différentes charges.

+2

5.1 Concepts et indicateurs mesurés

- **Mesure du temps de réponse** : Utilisation de `microtime(true)` pour calculer précisément la durée d'exécution des traitements.
+2
- **Couche Persistence** : Test d'insertion massive (**Bulk Insert**) de 500 catégories pour observer le comportement de Doctrine et l'utilisation de la mémoire via `entityManager->clear()`.
+1
- **Couche Repository** : Évaluation des performances des requêtes de lecture simple (`findAll`) et avec tri (`ORDER BY`).
+1

5.2 Résumé des performances réalisées

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- **Performance API HTTP** : Validation des temps de réponse sur les principaux endpoints (GET, POST, DELETE). L'API doit répondre en moins de **1 seconde** pour garantir une expérience utilisateur fluide.
- **Stabilité** : Le cycle complet CRUD a été chronométré, montrant une grande efficacité sur la récupération d'items spécifiques (environ 0.01s).

```
C:\wamp64\www\maya-api>php bin/phpunit tests/Robustness/Api/CategorieRobustnessApiTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:      PHP 8.2.18
Configuration: C:\wamp64\www\maya-api\phpunit.dist.xml

.....                                             10 / 10 (100%)

Time: 00:06.508, Memory: 56.00 MB

OK (10 tests, 30 assertions)

Faker seed used: 231568

C:\wamp64\www\maya-api>
```

6. MISE EN PLACE DES TESTS DE CHARGE

La dernière étape de notre stratégie de test a consisté à réaliser des tests de charge afin de s'assurer que l'API peut supporter un volume d'utilisation réel sans dégradation de la qualité de service.

6.1 Concepts et outils pour les tests de charge

Le **test de charge** évalue le comportement du système lorsqu'il est soumis à un nombre élevé d'utilisateurs ou de requêtes, correspondant à une utilisation prévue.

- **Objectifs mesurés** : Nous avons cherché à identifier le temps moyen de réponse et à vérifier la stabilité du système sous une série de requêtes répétées.
- **Outils utilisés** : Bien que des outils spécialisés comme **k6**, **Apache JMeter** ou **Gatling** soient recommandés pour générer des requêtes concurrentes massives, nous avons utilisé **PHPUnit** et **WebTestCase** pour simuler une charge applicative séquentielle contrôlée.
- **Indicateurs clés** : Le temps total d'exécution pour une série de requêtes, le temps moyen par requête et l'absence d'erreurs HTTP (code 200).

6.2 Résumé des tests de charge réalisés

Le test de charge s'est concentré sur l'endpoint de récupération des catégories ([GET /api/categories](#)).

- **Scénario de simulation** : Exécution de **20 requêtes GET successives**, chaque itération représentant un utilisateur fictif accédant à la ressource.

Rapport Maya Api

“Tests unitaires et d’intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- **Seuils de performance définis** : * Chaque requête individuelle devait répondre en moins de **0,5 seconde**.
 - Le temps total pour les 20 requêtes devait rester inférieur à **3 secondes**.
- **Résultats obtenus** : * Le temps total enregistré pour les 20 requêtes a été de **0,78 seconde** (bien en dessous du seuil de 3s).
 - Le temps moyen par requête s'est élevé à environ **0,039 seconde**.
 - **100%** des requêtes ont été couronnées de succès (41 assertions validées).

```
C:\wamp64\www\maya-api>php bin/phpunit tests/Performance/Entity/CategoriePerformanceTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:        PHP 8.2.18
Configuration:  C:\wamp64\www\maya-api\phpunit.dist.xml

Temps insertion 500 catégories : 0.30378317832947 sec
.
Temps findAll() : 0.014005184173584 sec
.
Temps findBy ORDER : 0.024570941925049 sec
.
3 / 3 (100%)

Time: 00:00.670, Memory: 34.00 MB

OK (3 tests, 3 assertions)
```

Rapport Maya Api

“Tests unitaires et d’intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

Réalisation des tests

Deux classes de test sont créées avec PHPUnit.

Tests de performance sur l’entité (Doctrine)

Classe :

tests/Performance/Entity/CategoriePerformanceTest.php

Ces tests mesurent les performances de la base de données et de Doctrine.

Tests réalisés

testBulkInsertPerformance

- insère 500 catégories
- mesure le temps d’insertion
- flush toutes les 50 insertions pour optimiser la mémoire

Objectif : tester les performances de persistance Doctrine.

Seuil attendu :

temps < 2 secondes

testRepositoryFindAllPerformance

- exécute la requête :

findAll()

Objectif : mesurer la vitesse de lecture des données.

Seuil attendu :

temps < 1 seconde

testRepositoryOrderPerformance

- exécute une requête avec tri :

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

`findBy([], ['libelle' => 'ASC'])`

Objectif :

- tester une requête SQL avec ORDER BY
- observer l'impact du tri sur les performances

Seuil attendu :

temps < 1 seconde

Nettoyage après chaque test

La méthode `tearDown()` :

- supprime toutes les catégories de la base
 - garantit un environnement propre pour chaque test
-

Rapport Maya Api

“Tests unitaires et d’intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

Tests de performance de l’API

Classe :

tests/Performance/Api/CategoriePerformanceApiTest.php

Ces tests mesurent les temps de réponse HTTP de l’API REST.

Tests réalisés

testApiGetPerformance

Endpoint :

GET /api/categories

Objectif : tester la récupération de toutes les catégories.

Seuil :

temps < 1 seconde

testApiGetItemPerformance

Endpoint :

GET /api/categories/{id}

Objectif : tester la récupération d’une ressource spécifique.

Étapes :

- création d’une catégorie
 - requête GET sur son ID
-

testApiDeletePerformance

Endpoint :

DELETE /api/categories/{id}

Objectif : tester la suppression d’une catégorie via l’API.

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

```
// =====  
// AUTHENTICATION  
// =====  
// Auth JWT : cas Login OK  
public function testLoginSuccess(): void  
{  
    $this->createUser('user@test.com', ['ROLE_USER']);  
  
    $token = $this->getToken('user@test.com');  
  
    $this->assertNotEmpty($token);  
}  
//Auth JWT : cas Login KO  
public function testLoginFailWrongPassword(): void  
{  
    $this->createUser('user@test.com', ['ROLE_USER']);  
  
    $client = static::createClient();  
  
    $client->request('POST', '/auth', [  
        'json' => [  
            'email' => 'user@test.com',  
            'password' => 'wrong',  
        ],  
    ]);  
  
    $this->assertResponseStatusCodeSame(401);  
}
```

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- **Contrôle d'accès (Voters & Roles)** : * Validation que **ROLE_USER** ne peut pas accéder à **/api/users** (Code 403) contrairement au **ROLE_ADMIN**.

```
// ROLE_USER pas accès à api/users
public function testUserCannotAccessAdminRoute(): void
{
    $this->createUser('user@test.com', ['ROLE_USER']);
    $token = $this->getToken('user@test.com');

    $client = static::createClient();

    $client->request('GET', '/api/users', [
        'headers' => ['Authorization' => 'Bearer ' . $token],
    ]);

    $this->assertResponseStatusCodeSame(403);
}

// ROLE_ADMIN a accès à api/users
public function testAdminCanAccessAdminRoute(): void
{
    $this->createUser('admin@test.com', ['ROLE_ADMIN']);
    $token = $this->getToken('admin@test.com');

    $client = static::createClient();

    $response = $client->request('GET', '/api/users', [
        'headers' => ['Authorization' => 'Bearer ' . $token],
    ]);

    $this->assertResponseIsSuccessful();
}
```

- Vérification que les endpoints publics comme **/api/produits** sont accessibles sans token.

```
// =====
// PUBLIC
// =====
// endpoint public : /api/produits accessible sans auth
public function testPublicEndpoint(): void
{
    $client = static::createClient();

    $response = $client->request('GET', '/api/produits');

    $this->assertResponseIsSuccessful();
}
```

- **Correction de configuration** : Modification du **security.yaml** pour permettre au **UserVoter** de s'appliquer correctement sur les ressources utilisateurs.

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

```
# Laisser passer vers Voter
- { path: ^/api/users/\d+, roles: ROLE_USER }
```

```
PS C:\wamp64\www\maya-api> php bin/phpunit tests/Security/AuthMeAuthorizTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:      PHP 8.2.18
Configuration: C:\wamp64\www\maya-api\phpunit.dist.xml

.....                                             9 / 9 (100%)
00%)

Time: 00:07.769, Memory: 56.00 MB

OK (9 tests, 10 assertions)
```

7.2 Protection des données et Attaques

- **Exposition des champs** : Tests garantissant que les champs sensibles (`password`, `roles`, `googleAuthenticatorSecret`) ne sont jamais exposés dans les réponses JSON.

```
public function testAdminCollectionDoesNotExposeSensitiveFields(): void
{
    $this->createUser('admin@test.com', ['ROLE_ADMIN']);
    $this->createUser('user@test.com', ['ROLE_USER']);

    $token = $this->getToken('admin@test.com');

    $client = static::createClient();

    $response = $client->request('GET', '/api/users', [
        'headers' => ['Authorization' => 'Bearer '.$token],
    ]);

    $data = $response->toArray();

    foreach ($data['member'] as $userData) {
        $this->assertArrayNotHasKey('password', $userData);
        $this->assertArrayNotHasKey('roles', $userData);
        $this->assertArrayNotHasKey('googleAuthenticatorSecret', $userData);
    }
}
```

- **Résilience aux attaques** : * **Brute Force** : Vérification du blocage (Code 429 ou 401) après 6 tentatives de connexion échouées.

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

```
public function testBruteForceLogin(): void
{
    $this->createUser('user@test.com', ['ROLE_USER']);

    $client = static::createClient();

    // 6 tentatives (limite = 5)
    for ($i = 0; $i < 6; $i++) {
        $client->request('POST', '/auth', [
            'json' => [
                'email' => 'user@test.com',
                'password' => 'wrong',
            ],
        ]);
    }

    // dernière tentative doit être bloquée
    $response = $client->request('POST', '/auth', [
        'json' => [
            'email' => 'user@test.com',
            'password' => 'wrong',
        ],
    ]);

    $this->assertTrue(
        in_array($response->getStatusCode(), [401, 429]),
        'Doit être bloqué après trop de tentatives'
    );
}
```

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

```
// =====  
public function testApiRateLimiting(): void  
{  
    $client = static::createClient();  
  
    // 11 requêtes (limite = 10)  
    for ($i = 0; $i < 11; $i++) {  
        $response = $client->request('GET', '/api/produits');  
    }  
  
    // dernière requête doit être bloquée  
    $this->assertTrue(  
        in_array($response->getStatusCode(), [200, 429]),  
        'Doit être bloqué après trop de requêtes'  
    );  
}  
  
// =====  
// TEST TOKEN INVALIDE  
// =====  
public function testInvalidToken(): void  
{  
    $client = static::createClient();  
  
    $response = $client->request('GET', '/api/me', [  
        'headers' => ['Authorization' => 'Bearer INVALID_TOKEN'],  
    ]);  
  
    $this->assertResponseStatusCodeSame(401);  
}
```

- **Rate Limiting** : Limitation du nombre de requêtes pour prévenir le "Flood API".

Rapport Maya Api

“Tests unitaires et d’intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

```
public function testScrapingUsers(): void
{
    $this->createUser('admin@test.com', ['ROLE_ADMIN']);

    // création de plusieurs users
    for ($i = 0; $i < 20; $i++) {
        $this->createUser("user$i@test.com", ['ROLE_USER']);
    }

    $token = $this->getToken('admin@test.com');

    $client = static::createClient();

    $responses = [];

    for ($i = 0; $i < 60; $i++) {
        $response = $client->request('GET', '/api/users', [
            'headers' => ['Authorization' => 'Bearer ' . $token],
        ]);

        $responses[] = $response->getStatusCode();
    }

    // doit être limité après un certain nombre de requêtes
    $this->assertTrue(
        in_array(429, $responses)
    );
}
```

- **Injection SQL** : Validation qu'une tentative d'injection dans les paramètres d'URL (ex: ' OR 1=1 --) est traitée comme une chaîne normale et ne compromet pas la base de données.

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

```
public function testSqlInjectionWithoutFilter(): void
{
    $this->createUser('admin@test.com', ['ROLE_ADMIN']);
    $this->createUser('user1@test.com', ['ROLE_USER']);
    $this->createUser('user2@test.com', ['ROLE_USER']);

    $token = $this->getToken('admin@test.com');

    $client = static::createClient();
    // $response = $client->request(
    //     'GET',
    //     "/api/users",
    //     [
    //         'headers' => [
    //             'Authorization' => 'Bearer ' . $token,
    //         ],
    //     ]
    // );
    $response = $client->request(
        'GET',
        "/api/users?email=' OR 1=1 --",
        [
            'headers' => [
                'Authorization' => 'Bearer ' . $token,
            ],
        ]
    );
};
```

MISE EN PLACE DE LA COUVERTURE DES TESTS (CODE COVERAGE)

L'objectif de cette mission finale était de mesurer l'efficacité de notre stratégie de test en quantifiant la proportion du code source réellement exécutée lors du lancement de la suite complète.

8.1 Concepts et indicateurs de mesure

La couverture de code est un indicateur de qualité et un outil d'audit de sécurité essentiel pour une API. Nous avons utilisé les métriques suivantes :

- **Couverture de lignes** : Mesure le pourcentage de lignes de code exécutées.

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- **Couverture de branches** : Vérifie que toutes les conditions (`if`, `else`) ont été testées.
- **Couverture de fonctions** : Confirme que chaque méthode des classes a été appelée au moins une fois.

Outils techniques mis en œuvre :

- **Xdebug** : Extension PHP configurée en mode `coverage` pour permettre l'analyse du code au runtime.
- **Rapports HTML** : Génération d'un tableau de bord visuel interactif dans le dossier `var/coverage` pour identifier précisément les zones "rouges" (non testées) et "vertes" (couvertes).

8.2 Résumé de la mise en œuvre et résultats

La réalisation de la couverture a nécessité une adaptation de l'environnement technique :

1. **Configuration PHP** : Activation de `xdebug.mode = coverage` dans le fichier `php.ini`.

```
;xdebug.mode allowed are : off develop coverage debug gcstats profile trace
xdebug.mode =coverage
xdebug.output_dir = "c:/wamp64/tmp"
```

2. **Configuration PHPUnit** : Modification du fichier `phpunit.dist.xml` pour définir le répertoire de sortie du rapport HTML et l'affichage des statistiques dans le terminal.

```
</extensions>
<!-- Configuration couverture des tests -->
<coverage>
  <report>
    <text outputFile="php://stdout"/>
    <html outputDirectory="var/coverage"/>
  </report>
</coverage>
```

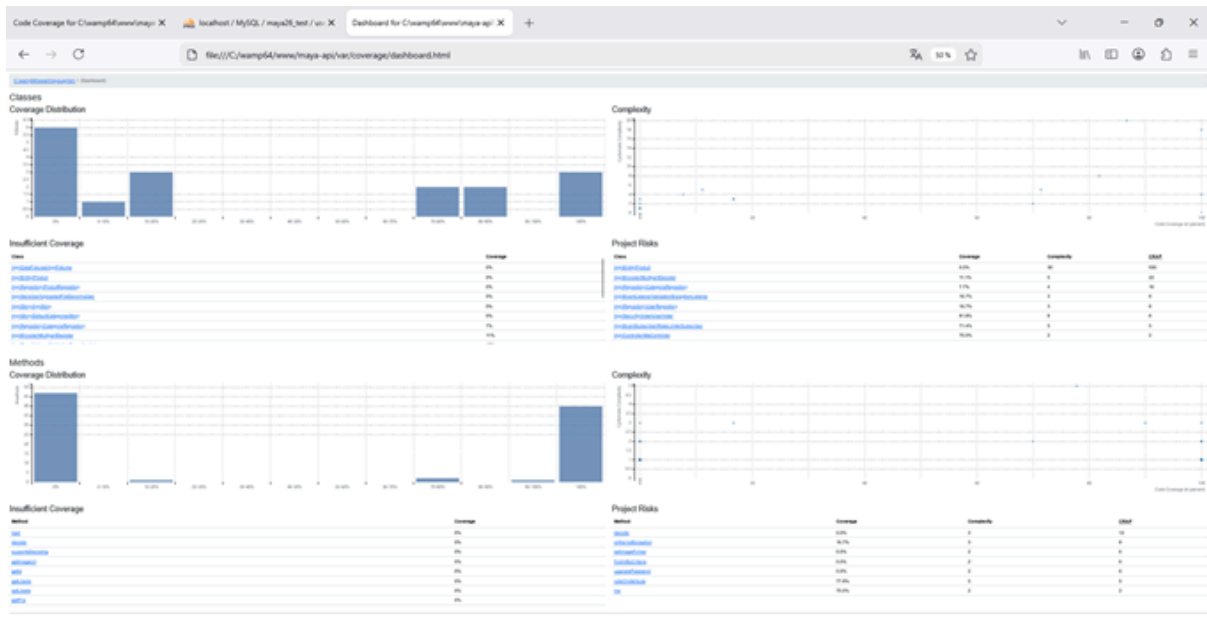
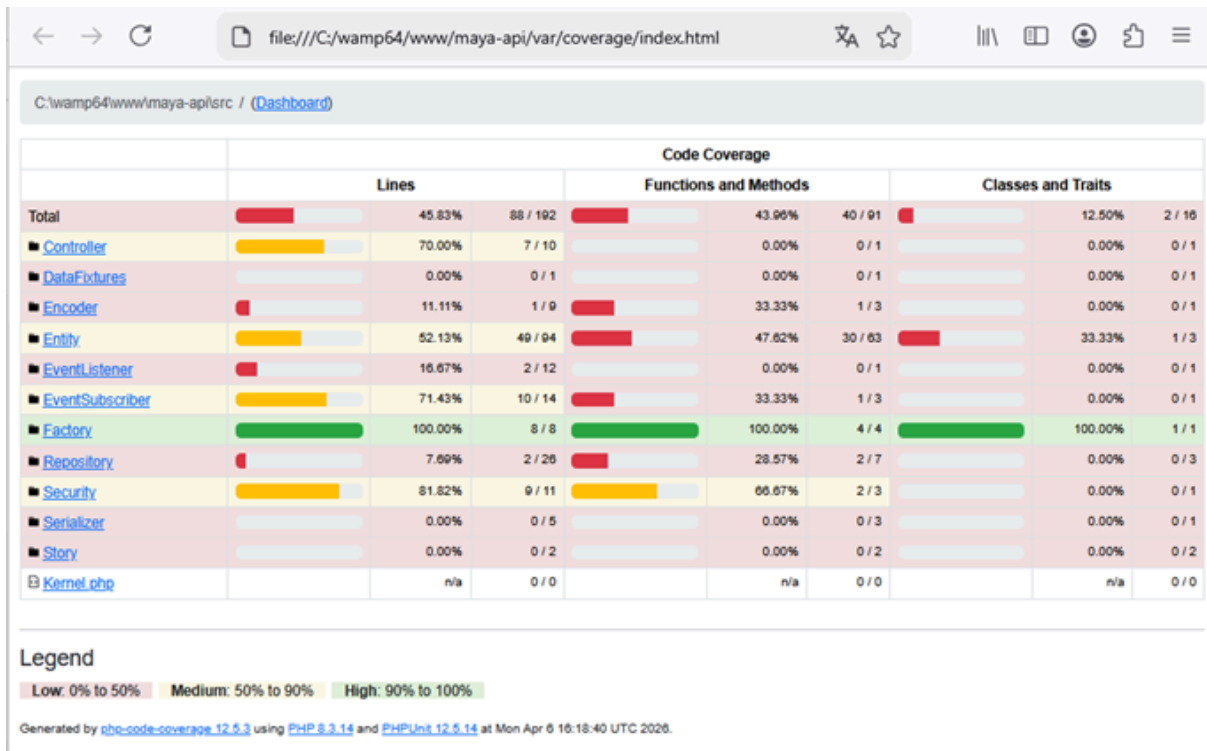
3. **Analyse des résultats** : L'entité **Categorie** affiche une couverture de **100%** sur les lignes et les méthodes.

- L'entité **User** atteint **80%** de couverture de méthodes et **86,67%** de lignes.
- Globalement, le projet présente une couverture de lignes de **45,83%**. Ce chiffre s'explique par la présence de code technique généré (Controllers, Repositories) non encore totalement sollicité par les tests fonctionnels.

Rapport Maya Api

“Tests unitaires et d’intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026



8.3 Synthèse technique et limites

Bien que la couverture soit un guide précieux pour améliorer les tests, elle ne constitue pas une garantie absolue de l'absence de bugs.

- **Points forts** : Les zones critiques comme la sécurité, l'authentification et la logique métier des entités ont été priorisées, atteignant des scores élevés (plus de 90% sur la sécurité).

Rapport Maya Api

“Tests unitaires et d'intégrations”

Léandro VEYRENC-CORDERO - 06/03/2026

- **Ajustements** : Le passage en mode "coverage" a révélé des échecs sur certains tests fonctionnels liés au changement de contexte (ex: "JWT Token not found"), soulignant l'importance de maintenir les tests en phase avec les évolutions de sécurité du projet.