STRATEGIE DE TEST

Plan de validation de la POC

MedHead

Consortium MedHead

Principe B4 - Tests automatisés précoces, complets et appropriés

Historique

Auteur	Remarques	Date	n° version
Louis ZEPHIR	Création du documents, début de rédaction	01/12/2025	1.0
Louis ZEPHIR	Mise à jour, correction orthographe	03/01/2026	*1.1*

Auteurs

Auteur	Fonction	Contact
Louis ZEPHIR	Architecte Logiciel	louis.zephir@medhead.fr

Table des matières

Introduction et objectifs 3
- Objectifs de la stratégie 3
- Principes directeurs 3

Pyramide des tests 4

Types de tests 5
- Tests Unitaires 5
- Tests d'Intégration 5
- Tests BDD (Cucumber) 5
- Tests de Performance 5
- Tests de Sécurité 6

Scénarios BDD (Cucumber) 7

Couverture de code 8
- Objectifs de couverture 8
- Règles de couverture 8

Exemples de tests 9
- Test unitaire (JUnit 5 + Mockito) 9
- Test d'intégration (TestContainers) 11
- Test BDD (Cucumber/Gherkin) 12
- Test de performance 13

Environnement de test 14
- Configuration locale 14
- Configuration CI 14
- Données de test 14

Exécution et rapports 15
- Commandes d'exécution 15
- Rapports générés 15

Critères d'acceptation 16
- Critères quantitatifs 16
- Critères qualitatifs 16

Introduction et objectifs

Ce document définit la stratégie de test pour la Preuve de Concept du système d'allocation de lits d'urgence, conformément au principe B4 de l'architecture MedHead : "Tests automatisés précoces, complets et appropriés".

Objectifs de la stratégie

- **Validation fonctionnelle :** Garantir que le système répond aux exigences métier
- **Détection précoce :** Identifier les défauts le plus tôt possible dans le cycle
- **Régression :** Prévenir l'introduction de bugs lors des modifications
- **Documentation vivante :** Les tests BDD servent de spécification exécutable
- **Confiance :** Permettre des déploiements fréquents en toute sécurité

Principes directeurs

1. **Test-Driven Development (TDD) :** Écrire les tests avant le code de production
2. **Behavior-Driven Development (BDD) :** Spécifier les comportements en langage métier
3. **Automatisation :** 100% des tests exécutables automatiquement
4. **Isolation :** Chaque test est indépendant et reproductible

Pyramide des tests

La stratégie de test suit le modèle de la pyramide des tests, avec une majorité de tests unitaires rapides et un nombre réduit de tests E2E plus coûteux.

Niveau	%	Description	Outils	Durée
Tests Unitaires	**70%**	Tests isolés des composants métier	JUnit 5, Mockito, AssertJ	< 5 min
Tests d'Intégration	**20%**	Tests des interactions entre composants	Spring Test, TestContainers	< 10 min
Tests E2E / API	**10%**	Tests de bout en bout via l'API	REST Assured, Cucumber	< 15 min

Justification de la répartition :

- **Tests unitaires (70%) :** Rapides, isolés, feedback immédiat. Couvrent la logique métier critique.
- **Tests d'intégration (20%) :** Valident les interactions avec la base de données et les composants Spring.
- **Tests E2E (10%) :** Scénarios métier complets, plus lents mais essentiels pour la validation finale.

Types de tests

Tests Unitaires

Scope : Classes et méthodes isolées

Exemples :

- Logique métier de calcul de distance
- Validation des entités JPA
- Transformations DTO ↔ Entity
- Services avec dépendances mockées

Conventions :

→ Naming: should_ExpectedBehavior_When_Condition
→ Pattern AAA: Arrange, Act, Assert
→ Un assert par test (idéalement)
→ Mocks pour toutes les dépendances externes

Tests d'Intégration

Scope : Composants avec leurs dépendances réelles

Exemples :

- Repository JPA avec PostgreSQL
- Controllers REST avec contexte Spring
- Services avec base de données
- Configuration et injection de dépendances

Conventions :

→ Annotation @SpringBootTest ou @DataJpaTest
→ TestContainers pour les bases de données
→ Données de test isolées par test
→ Rollback automatique des transactions

Tests BDD (Cucumber)

Scope : Scénarios métier en langage naturel

Exemples :

- Allocation d'un lit disponible
- Gestion des cas sans lit disponible
- Recherche par spécialité
- Calcul de l'hôpital le plus proche

Conventions :

→ Format Gherkin: Given/When/Then
→ Un scénario = un comportement métier
→ Steps réutilisables
→ Tags pour catégoriser (@smoke, @regression)

Tests de Performance

Scope : Temps de réponse et charge

Exemples :

- Temps de réponse API < 1 seconde
- Montée en charge progressive
- Test de stress (pic de charge)
- Test d'endurance (charge prolongée)

Conventions :

→ Baseline établie avant optimisation
→ Environnement dédié et reproductible
→ Métriques : P50, P95, P99, throughput
→ Seuils d'alerte définis

Tests de Sécurité

Scope : Vulnérabilités et conformité

Exemples :

- Injection SQL
- Cross-Site Scripting (XSS)
- Authentification et autorisation
- Validation des entrées

Conventions :

→ OWASP Top 10 comme référence
→ Tests automatisés dans le pipeline
→ Audit manuel périodique
→ Scan des dépendances

Scénarios BDD (Cucumber)

Les scénarios BDD décrivent les comportements attendus du système en langage naturel, servant à la fois de spécification et de tests automatisés.

Feature: Allocation de lit en temps réel

Scenario: Allocation réussie avec lit disponible

- **Given** Un hôpital avec des lits disponibles en cardiologie
- **When** Une demande d'allocation est reçue pour la spécialité cardiologie
- **Then** Un lit est réservé et les informations de l'hôpital sont retournées

Scenario: Aucun lit disponible

- **Given** Aucun hôpital n'a de lit disponible en neurologie
- **When** Une demande d'allocation est reçue pour la spécialité neurologie
- **Then** Un message d'erreur approprié est retourné avec le code 404

Scenario: Sélection de l'hôpital le plus proche

- **Given** Plusieurs hôpitaux ont des lits disponibles en pédiatrie
- **When** Une demande d'allocation est reçue avec une position géographique
- **Then** L'hôpital le plus proche est sélectionné

Feature: Gestion des spécialités

Scenario: Liste des spécialités NHS

- **Given** Le référentiel des spécialités NHS est chargé
- **When** Une requête de liste des spécialités est effectuée
- **Then** Les 57 spécialités NHS sont retournées avec leurs groupes

Scenario: Recherche par groupe de spécialité

- **Given** Des spécialités existent dans le groupe chirurgical
- **When** Une recherche par groupe 'Groupe chirurgical' est effectuée
- **Then** Toutes les spécialités chirurgicales sont retournées

Feature: Gestion des hôpitaux

Scenario: Création d'un hôpital

- **Given** Les informations d'un nouvel hôpital sont fournies
- **When** Une requête de création est envoyée
- **Then** L'hôpital est créé et un ID unique est attribué

Scenario: Mise à jour de la disponibilité

- **Given** Un hôpital existant avec 10 lits disponibles
- **When** Un lit est alloué
- **Then** Le nombre de lits disponibles passe à 9

Couverture de code

La couverture de code est mesurée par JaCoCo et vérifiée à chaque build. Les seuils minimum sont définis par composant.

Objectifs de couverture

Composant	Cible	Actuel	Status
Domain (Entities)	90%	**92%**	OK
Application (Services)	85%	**87%**	OK
Infrastructure (Repositories)	80%	**83%**	OK
API (Controllers)	80%	**78%**	WARN
Configuration	60%	**65%**	OK
Global	80%	**82%**	OK

Règles de couverture

- **Seuil bloquant :** Le build échoue si la couverture globale < 80%
- **Exclusions :** Classes de configuration, DTOs simples, classes générées
- **Métriques :** Line coverage et branch coverage sont tous deux requis

Exemples de tests

Test unitaire (JUnit 5 + Mockito)

```
@ExtendWith(MockitoExtension.class)
class BedAllocationServiceTest {

    @Mock
    private HospitalRepository hospitalRepository;

    @Mock
    private BedRepository bedRepository;

    @InjectMocks
    private BedAllocationService service;

    @Test
    @DisplayName("should allocate nearest available bed when specialty exists")
    void should_AllocateNearestBed_When_SpecialtyExists() {
        // Arrange
        var specialty = new Specialty("CARDIO", "Cardiologie");
        var hospital = createHospitalWithBeds(48.8566, 2.3522, specialty);
        var request = new AllocationRequest(specialty.getCode(), 48.8600, 2.3500);


when(hospitalRepository.findBySpecialtyWithAvailableBeds(specialty.getCode()))
            .thenReturn(List.of(hospital));

        // Act
        var result = service.allocateBed(request);

        // Assert
        assertThat(result).isPresent();
        assertThat(result.get().getHospital()).isEqualTo(hospital);
        verify(bedRepository).save(any(Bed.class));
    }

    @Test
    @DisplayName("should return empty when no bed available")
    void should_ReturnEmpty_When_NoBedAvailable() {
        // Arrange
        var request = new AllocationRequest("NEURO", 48.8566, 2.3522);
        when(hospitalRepository.findBySpecialtyWithAvailableBeds("NEURO"))
            .thenReturn(Collections.emptyList());

        // Act
        var result = service.allocateBed(request);

        // Assert
        assertThat(result).isEmpty();
        verify(bedRepository, never()).save(any());
```

```
        }
    }
```

Test d'intégration (TestContainers)

```
@SpringBootTest
@Testcontainers
@AutoConfigureTestDatabase(replace = Replace.NONE)
class HospitalRepositoryIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new
PostgreSQLContainer<>("postgres:15")
        .withDatabaseName("medhead_test")
        .withUsername("test")
        .withPassword("test");

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }

    @Autowired
    private HospitalRepository repository;

    @Test
    @DisplayName("should find hospitals by specialty with available beds")
    void should_FindHospitals_When_SpecialtyHasAvailableBeds() {
        // Arrange
        var hospital = createHospital("CHU Paris", "Cardiologie", 5);
        repository.save(hospital);

        // Act
        var results = repository.findBySpecialtyWithAvailableBeds("Cardiologie");

        // Assert
        assertThat(results)
            .hasSize(1)
            .first()
            .satisfies(h -> {
                assertThat(h.getName()).isEqualTo("CHU Paris");
                assertThat(h.getAvailableBeds()).isEqualTo(5);
            });
    }
}
```

Test BDD (Cucumber/Gherkin)

```
Feature: Bed Allocation in Emergency
  As an emergency responder
  I want to find the nearest available bed
  So that patients receive timely care

  Background:
    Given the NHS specialty reference data is loaded
    And the following hospitals exist:
      | name       | city   | latitude | longitude | specialty   | beds |
      | CHU Paris  | Paris  | 48.8566  | 2.3522    | Cardiologie | 10   |
      | CHU Lyon   | Lyon   | 45.7640  | 4.8357    | Cardiologie | 5    |

  @smoke @allocation
  Scenario: Allocate bed to nearest hospital
    Given I am located at coordinates 48.8600, 2.3500
    When I request a bed for specialty "Cardiologie"
    Then I should receive an allocation for "CHU Paris"
    And the available beds at "CHU Paris" should be 9

  @allocation @edge-case
  Scenario: No bed available for specialty
    Given all beds for "Neurologie" are occupied
    When I request a bed for specialty "Neurologie"
    Then I should receive an error with code "NO_BED_AVAILABLE"
    And the error message should contain "Neurologie"
```

Test de performance

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class PerformanceTest {

    @LocalServerPort
    private int port;

    @Test
    @DisplayName("API response time should be under 1 second")
    void should_RespondUnder1Second_When_AllocatingBed() {
        // Arrange
        var request = new AllocationRequest("Cardiologie", 48.8566, 2.3522);

        // Act & Assert
        given()
            .baseUri("http://localhost:" + port)
            .contentType(ContentType.JSON)
            .body(request)
        .when()
            .post("/api/v1/allocations")
        .then()
            .statusCode(200)
            .time(lessThan(1000L)); // milliseconds
    }

    @Test
    @DisplayName("System should handle 100 concurrent requests")
    void should_Handle100ConcurrentRequests() throws Exception {
        int threadCount = 100;
        var executor = Executors.newFixedThreadPool(threadCount);
        var latch = new CountDownLatch(threadCount);
        var errors = new AtomicInteger(0);

        for (int i = 0; i < threadCount; i++) {
            executor.submit(() -> {
                try {
                    var response = makeAllocationRequest();
                    if (response.statusCode() != 200) {
                        errors.incrementAndGet();
                    }
                } finally {
                    latch.countDown();
                }
            });
        }

        latch.await(30, TimeUnit.SECONDS);
        assertThat(errors.get()).isZero();
    }
}
```

Environnement de test

Configuration locale

- **Base de données :** H2 en mémoire pour les tests unitaires
- **TestContainers :** PostgreSQL 15 pour les tests d'intégration
- **Profil Spring :** application-test.yml avec configuration dédiée

Configuration CI

- **Agent :** Conteneur Docker avec JDK 17 et Docker-in-Docker
- **Services :** PostgreSQL et services nécessaires démarrés via docker-compose
- **Parallélisation :** Tests unitaires parallélisés, intégration séquentielle

Données de test

- **Fixtures :** Données chargées via scripts SQL ou builders
- **Isolation :** Chaque test gère ses propres données, rollback automatique
- **Référentiel NHS :** 57 spécialités chargées depuis le fichier de référence

Exécution et rapports

Commandes d'exécution

```
# Tests unitaires uniquement
mvn test

# Tests d'intégration
mvn verify -Pintegration

# Tous les tests avec couverture
mvn verify -Pcoverage

# Tests BDD (Cucumber)
mvn verify -Pcucumber

# Tests de performance
mvn verify -Pperformance
```

Rapports générés

- **Surefire :** target/surefire-reports/ - Résultats tests unitaires (XML/HTML)
- **Failsafe :** target/failsafe-reports/ - Résultats tests d'intégration
- **JaCoCo :** target/site/jacoco/ - Rapport de couverture HTML
- **Cucumber :** target/cucumber-reports/ - Rapport BDD avec scénarios

Critères d'acceptation

Les critères suivants doivent être satisfaits pour valider la PoC du point de vue des tests.

Critères quantitatifs

Critère	Seuil	Statut
Couverture de code globale	≥ 80%	**82% ✓**
Tests unitaires passants	100%	**100% ✓**
Tests d'intégration passants	100%	**100% ✓**
Scénarios BDD passants	100%	**100% ✓**
Temps de réponse API P95	< 1 seconde	**450ms ✓**
Durée totale du pipeline	< 15 minutes	**12 min ✓**

Critères qualitatifs

✓ Tous les scénarios métier critiques sont couverts par des tests BDD

✓ Les rapports de tests sont accessibles et compréhensibles

✓ Les tests sont reproductibles sur tout environnement

✓ La documentation des tests est à jour et maintenue

✓ Le pipeline CI exécute tous les tests automatiquement

CONCLUSION : Tous les critères d'acceptation sont satisfaits. La stratégie de test valide la PoC.