

# Jeux de tests unitaires et d'intégration

Ce document présente la stratégie de tests mise en place dans les services du projet, avec un exemple détaillé basé sur `penpal-ai-db-service`. Il couvre les tests unitaires, les tests d'intégration (E2E), et les bonnes pratiques de couverture de code.

## Vue d'ensemble

Chaque service du projet dispose d'une suite complète de tests :

- **Tests unitaires** : Validation isolée des services, contrôleurs et utilitaires
- **Tests E2E** : Validation des workflows complets via API HTTP
- **Tests de couverture** : Mesure de la couverture de code avec seuils minimum
- **Tests d'intégration** : Validation des interactions entre composants

## Architecture de tests

```
# Structure type d'un service
penpal-ai-*-service/
├── src/
│   ├── modules/
│   │   └── users/
│   │       ├── users.service.ts
│   │       ├── users.service.spec.ts      # Tests unitaires
│   │       ├── users.controller.ts
│   │       └── users.controller.spec.ts    # Tests unitaires
│   └── test/
│       ├── users.e2e-spec.ts              # Tests E2E
│       ├── app.e2e-spec.ts                # Tests E2E globaux
│       ├── jest-e2e.json                  # Configuration E2E
│       └── package.json                    # Scripts de test
```

## Exemple détaillé : Tests du UserService

### 1. Service testé : UserService

Le `UserService` de `penpal-ai-db-service` gère les opérations CRUD des utilisateurs avec MongoDB et inclut des fonctionnalités métier complexes comme l'onboarding et les métriques.

```
async findOne(id: string): Promise<UserDocument> {
  try {
    const user = await this.userModel.findById(id).exec();
    if (!user) {
      throw new NotFoundException(`User with ID ${id} not found`);
    }
    return user;
  }
}
```

```

    catch (error) {
      if (error instanceof NotFoundException) {
        throw error;
      }
      this.logger.error(`Error finding user: ${error.message}`,
error.stack);
      throw new InternalServerErrorException("Failed to retrieve user");
    }
  }
}

```

## 2. Configuration du test unitaire

```

beforeEach(async () => {
  const mockModel = () => ({
    find: jest.fn().mockReturnThis(),
    skip: jest.fn().mockReturnThis(),
    limit: jest.fn().mockReturnThis(),
    exec: jest.fn().mockResolvedValue([]),
    findById: jest.fn().mockReturnThis(),
    findByIdAndUpdate: jest.fn().mockReturnThis(),
    findByIdAndDelete: jest.fn().mockReturnThis(),
    findOne: jest.fn().mockReturnThis(),
    countDocuments: jest.fn().mockReturnThis(),
    populate: jest.fn().mockReturnThis(),
    save: jest.fn(),
    deleteOne: jest.fn().mockReturnThis(),
  });

  const module: TestingModule = await Test.createTestingModule({
    providers: [
      UserService,
      { provide: getModelToken("User"), useFactory: mockModel },
      { provide: getModelToken("UserRole"), useFactory: mockModel },
      { provide: Logger, useValue: new Logger() },
    ],
  }).compile();

  service = module.get<UserService>(UserService);
  userModel = module.get(getModelToken("User")) as any;
  userRoleModel = module.get(getModelToken("UserRole")) as any;
});

```

### Explication de la configuration :

- **TestingModule** : Module de test NestJS isolé
- **Mocking des dépendances** : MongoDB models (**User**, **UserRole**) et **Logger** sont mockés
- **Injection des tokens** : **getModelToken()** pour récupérer les modèles Mongoose
- **Chaînage fluide** : Les méthodes mockées retournent **this** pour permettre **.findById().exec()**




## 3. Test unitaire détaillé - Cas d'erreur

```
it("findOne throws NotFoundException when not found", async () => {
  userModel.exec.mockResolvedValueOnce(null);
  await expect(service.findOne("nope")).rejects.toBeInstanceOf(
    NotFoundException,
  );
});
```

#### Analyse du test :

1. **Arrange** : Configure le mock pour retourner `null` (utilisateur inexistant)
2. **Act** : Appelle `findOne()` avec un ID qui n'existe pas
3. **Assert** : Vérifie qu'une `NotFoundException` est bien lancée

#### Couverture :

-  Branche d'erreur "utilisateur non trouvé"
-  Gestion des exceptions métier
-  Validation du type d'exception correct

#### 4. Test unitaire détaillé - Cas de succès avec logique métier

```
it("getOnboardingStatus returns needsOnboarding based on user doc",
  async () => {
    userModel.exec.mockResolvedValueOnce({ onboardingCompleted: false });
    const res = await service.getOnboardingStatus("u1");
    expect(res.needsOnboarding).toBe(true);
  });
```

#### Analyse du test :

1. **Mock de données** : Simule un utilisateur avec `onboardingCompleted: false`
2. **Logique métier** : Teste la transformation `!onboardingCompleted → needsOnboarding: true`
3. **Assertion spécifique** : Vérifie la propriété calculée, pas seulement le retour brut

#### 5. Test avec gestion d'erreur business

```
it("assignRole throws ConflictException when already exists", async ()
=> {
  userRoleModel.exec.mockResolvedValueOnce({ id: "r1" });
  await expect(service.assignRole("u", "r")).rejects.toBeInstanceOf(
    ConflictException,
  );
});
```

#### Couverture des règles métier :

- Teste la validation business "un utilisateur ne peut pas avoir deux fois le même rôle"
- Vérifie le type d'exception spécifique aux règles métier
- Simule l'état de base de données où le rôle existe déjà

## Tests d'intégration (E2E)

### Configuration E2E

```
beforeAll(async () => {
  const module: TestingModule = await Test.createTestingModule({
    controllers: [UsersController],
    providers: [
      {
        provide: UserService,
        useValue: {
          findAll: jest.fn().mockResolvedValue([mockUser]),
          findOne: jest.fn().mockResolvedValue(mockUser),
          findByEmail: jest.fn().mockResolvedValue(mockUser),
          getUserMetrics: jest.fn().mockResolvedValue({
            activeUsers: 1,
            totalUsers: 1,
            usersByLanguage: { en: 1 },
            averageUserLevel: { en: 2 },
          }),
          update: jest.fn().mockResolvedValue(mockUser),
          remove: jest.fn().mockResolvedValue(undefined),
          updateOnboardingProgress:
            jest.fn().mockResolvedValue(mockUser),
          completeOnboarding: jest.fn().mockResolvedValue(mockUser),
          getOnboardingStatus: jest
            .fn()
            .mockResolvedValue({ needsOnboarding: false }),
          create: jest.fn().mockResolvedValue(mockUser),
        },
      },
      { provide: Logger, useValue: new Logger("UsersControllerTest") },
      // Provide the guard used by the controller decorator as a
      // permissive mock
      { provide: ServiceAuthGuard, useValue: { canActivate: () => true } },
    ],
    // Provide ConfigService stub in case anything else needs it
    { provide: ConfigService, useValue: { get: jest.fn(() => "") } },
    {
      provide: CACHE_MANAGER,
      useValue: { get: jest.fn(), set: jest.fn(), del: jest.fn() },
    },
  ],
  {
    compile: true,
  }
}).compile();





app = module.createNestApplication();
process.env.NODE_ENV = "test";
await app.init();
```

```
usersService = module.get(UserService) as jest.Mocked<UserService>;
});
```

## Test E2E complet - Workflow API

```
it("GET /users returns list", async () => {
  const res = await
request(app.getHttpServer()).get("/users").expect(200);
  expect(res.body).toEqual([expect.objectContaining({ _id: "u1" })]);
  expect(usersService.findAll).toHaveBeenCalled();
});
```

### Couverture E2E :

-  **HTTP Layer** : Requête GET complète avec status code
-  **Controller Layer** : Vérification de l'appel au service
-  **Response Format** : Validation de la structure JSON retournée
-  **Authentication** : Guard d'authentification mocké

## Test E2E - Workflow complexe avec payload

```
it("PATCH /users/:id/onboarding/progress saves progress", async () => {
  const res = await request(app.getHttpServer())
    .patch("/users/u1/onboarding/progress")
    .send({ currentStep: "profile" })
    .expect(200);
  expect(res.body._id).toBe("u1");

  expect(usersService.updateOnboardingProgress).toHaveBeenCalledWith("u1", {
    currentStep: "profile" });
});
```

### Validation workflow :

- **Request Mapping** : Paramètre URL + body JSON
- **Service Integration** : Vérification des arguments transmis
- **Response Validation** : Contenu et format de réponse

## Scripts de test et couverture

### Configuration des scripts

```
"test": "jest",
"test:watch": "jest --watch",
"test:cov": "jest --coverage",
"test:debug": "node --inspect-brk -r tsconfig-paths/register -r ts-
```

```
node/register node_modules/.bin/jest --runInBand",
  "test:e2e": "jest --config ./test/jest-e2e.json",
```

## Exécution des tests

```
# Tests unitaires
npm test

# Tests unitaires avec surveillance continue
npm run test:watch

# Tests unitaires avec couverture
npm run test:cov

# Tests E2E
npm run test:e2e

# Tests avec debugging
npm run test:debug

# Séquence complète (comme en CI)
npm run lint
npm test -- --runInBand
npm run test:cov -- --runInBand
npm run test:e2e -- --runInBand
```

## Couverture de code

La configuration Jest assure une couverture minimale :

```
{
  "collectCoverageFrom": [
    "src/**/*.ts",
    "!src/main.ts",
    "!src/**/*.interface.ts",
    "!**/*.spec.ts",
    "!**/*.e2e-spec.ts"
  ],
  "coverageThreshold": {
    "global": {
      "branches": 80,
      "functions": 80,
      "lines": 80,
      "statements": 80
    }
  }
}
```

# Bonnes pratiques mises en œuvre

## 1. Isolation des tests

- **Mocking des dépendances** : MongoDB, services externes, logging
- **Tests hermétiques** : Chaque test est indépendant
- **Environnement contrôlé** : Variables d'environnement dédiées au test

## 2. Patterns de test

- **AAA Pattern** : Arrange, Act, Assert clairement séparés
- **Descriptive naming** : Noms de tests explicites sur le comportement testé
- **Edge cases** : Tests des cas d'erreur et limites
- **Business logic** : Validation des règles métier spécifiques

## 3. Couverture fonctionnelle

- **Happy path** : Scénarios de succès
- **Error path** : Gestion d'erreurs et exceptions
- **Boundary conditions** : Valeurs limites et cas extrêmes
- **Integration points** : Interactions entre composants

## 4. Maintenance des tests

- **Mock factories** : Réutilisation de mocks cohérents
- **Test utilities** : Helpers pour setup récurrent
- **Clear teardown** : Nettoyage après tests
- **Fast feedback** : Tests rapides pour développement itératif

# Tests disponibles par service

Tous les services du projet suivent cette même architecture de tests :

## Services backend (NestJS)

- **penpal-ai-db-service** : ✅ Tests unitaires + E2E
- **penpal-ai-auth-service** : ✅ Tests unitaires + E2E
- **penpal-ai-asimov-service** : ✅ Tests unitaires + E2E
- **payment-service** : ✅ Tests unitaires + E2E
- **penpal-ai-notify-service** : ✅ Tests unitaires + E2E
- **penpal-ai-monitoring-service** : ✅ Tests unitaires + E2E

## Service frontend (Next.js)

- **penpal-frontend** : ✅ Tests unitaires (Jest) + Tests E2E (Cypress)

## Commandes uniformes

Chaque service expose les mêmes commandes :

```
# Dans chaque service
npm test                # Tests unitaires
npm run test:cov        # Couverture
npm run test:e2e        # Tests E2E
npm run test:watch      # Mode surveillance
```

## CI/CD Integration

Tous les tests sont exécutés automatiquement dans le pipeline CI :

```
# Séquence CI pour chaque service
npm run lint
npm test -- --runInBand
npm run test:cov -- --runInBand
npm run test:e2e -- --runInBand
```

## Outils et frameworks

### Backend (NestJS)

- **Jest** : Framework de test principal
- **Supertest** : Tests HTTP E2E
- **@nestjs/testing** : Utilitaires de test NestJS
- **Mocking** : Jest mocks pour dépendances externes

### Frontend (Next.js)

- **Jest** : Tests unitaires et composants
- **React Testing Library** : Tests de composants React
- **Cypress** : Tests E2E interface utilisateur
- **Mock Service Worker** : Mocking d'API pour tests

### Métriques de qualité

- **Couverture** : Minimum 80% (branches, fonctions, lignes, statements)
- **Performance** : Tests rapides (< 30s par service)
- **Fiabilité** : Tests déterministes, pas de flaky tests
- **Maintenabilité** : Tests lisibles et faciles à modifier

## Stratégie de debugging

### Tests unitaires failing

```
# Mode debug avec breakpoints
npm run test:debug
```



```
# Tests spécifiques avec output détaillé  
npm test -- --verbose users.service.spec.ts  
  
# Mode surveillance pour développement itératif  
npm run test:watch -- users.service.spec.ts
```

## Tests E2E failing

```
# E2E avec logs détaillés  
npm run test:e2e -- --verbose  
  
# Variables d'environnement de debug  
DEBUG=* npm run test:e2e
```

Cette architecture de tests garantit la fiabilité, la maintenabilité et la qualité du code sur l'ensemble des services du projet, avec une couverture fonctionnelle complète et des pratiques uniformes.