# 1. Introduction aux Tests

Les tests sont essentiels pour garantir la qualité et la fiabilité d'une application. Ils permettent de valider le bon fonctionnement du code, de prévenir les régressions, et de faciliter la maintenance. En développement, il existe plusieurs types de tests, chacun répondant à un besoin spécifique. Voyons-les en détail.

---

# 2. Types de Tests

## 2.1 Tests Unitaires

**Explication**

Les tests unitaires vérifient de manière isolée le comportement de petites unités de code, généralement des fonctions ou des méthodes. Ces tests sont rapides, indépendants des autres tests, et se concentrent sur la logique de chaque unité de code.

**Exemple en JavaScript avec Jest**

Jest est un framework de test JavaScript populaire pour les tests unitaires. Voici un exemple :

```
// math.js
function add(a, b) {
  return a + b;
}
module.exports = add;

// math.test.js
const add = require('./math');

test('ajoute 1 + 2 pour obtenir 3', () => {
  expect(add(1, 2)).toBe(3);
});
```

Pour exécuter ce test avec Jest :

```
npm install --save-dev jest
npx jest
```

**Documentation : Jest - Getting Started**

---

## 2.2 Tests d'Intégration

**Explication**

Les tests d'intégration vérifient comment plusieurs modules ou composants interagissent entre eux. Ces tests sont plus larges que les tests unitaires car ils valident des ensembles de fonctions ou de composants pour s'assurer qu'ils fonctionnent ensemble comme prévu.

**Exemple en JavaScript avec Mocha et Chai**

Mocha est un framework de test JavaScript, et Chai est une bibliothèque d'assertions.

```
// user.js
function authenticate(username, password) {
  if (username === 'admin' && password === '1234') return 'token';
  return null;
}
module.exports = authenticate;

// user.test.js
const chai = require('chai');
const expect = chai.expect;
const authenticate = require('./user');

describe('Authentification', () => {
  it('retourne un token pour des identifiants valides', () => {
    expect(authenticate('admin', '1234')).to.equal('token');
  });

  it("retourne null pour des identifiants invalides", () => {
    expect(authenticate('user', 'wrongpassword')).to.be.null;
  });
});
```

Pour exécuter ce test :

```
npm install --save-dev mocha chai
npx mocha
```

**Documentation : Mocha - Getting Started**

---

## 2.3 Tests de Bout en Bout (End-to-End ou E2E)

**Explication**

Les tests E2E simulent un utilisateur interagissant avec l'application complète. Ils vérifient si toutes les parties d'un système fonctionnent correctement ensemble en passant par toute la chaîne de bout en bout. Ces tests sont plus lents mais permettent de s'assurer que toutes les intégrations fonctionnent correctement dans un environnement similaire à celui des utilisateurs finaux.

**Exemple en JavaScript avec Cypress**

Cypress est un framework populaire pour les tests E2E.

```
// test.cy.js
describe('Test de connexion', () => {
  it('doit se connecter avec des identifiants valides', () => {
    cy.visit('http://localhost:3000/login');
    cy.get('#username').type('admin');
    cy.get('#password').type('1234');
    cy.get('#login-button').click();
    cy.url().should('include', '/dashboard');
  });
});
```

Pour exécuter le test avec Cypress :

```
npm install --save-dev cypress
npx cypress open
```

**Documentation : Cypress - Getting Started**

---

### 2.4 Tests de Régression

**Explication**

Les tests de régression s'assurent qu'un changement de code (nouvelle fonctionnalité, bug fix, etc.) n'introduit pas de régressions, c'est-à-dire de nouvelles erreurs dans des fonctionnalités déjà existantes. Cela se fait souvent en réexécutant les tests unitaires, d'intégration et E2E après chaque modification majeure.

**Exemple en JavaScript**

Les tests de régression ne nécessitent pas de code spécifique, mais il est courant d'utiliser des solutions d'automatisation pour exécuter les tests de manière continue. **Jenkins** ou **GitHub Actions** sont couramment utilisés pour cela.

**Documentation : GitHub Actions**

---

## 3. Plans de Tests

Un plan de test est un document qui décrit la stratégie de test, les objectifs, les ressources nécessaires et les cas de test pour un projet. Il définit également les critères de succès ou d'échec des tests.

### Exemple de plan de test

Un plan de test pourrait inclure :

- **Objectifs du test** : S'assurer que l'application fonctionne selon les spécifications.
- **Types de tests** : Unitaire, Intégration, E2E, Régression.
- **Cas de test** : Liste des fonctionnalités à tester avec les entrées et les résultats attendus.
- **Critères de réussite/échec** : 95 % de couverture de test, aucun bug critique restant, etc.

---

## 4. Couverture de Test

La couverture de test mesure le pourcentage de code couvert par les tests. Elle permet de savoir si des parties du code n'ont pas encore été testées. Une couverture de test élevée ne garantit pas que le code est exempt de bugs, mais elle augmente la probabilité d'avoir détecté les défauts majeurs.

### Exemple de calcul de la couverture avec Jest

Jest permet de générer un rapport de couverture :

```
npx jest --coverage
```

Cela produira un rapport montrant les fichiers et les lignes de code testés, souvent sous forme de pourcentages (statements, branches, functions, lines).

**Documentation sur la couverture : Jest - Coverage**

En résumé, chaque type de test a un rôle spécifique :

- **Tests unitaires** : Vérifient les plus petites unités de code.
- **Tests d'intégration** : Vérifient l'interaction entre les composants.
- **Tests E2E** : Vérifient le parcours utilisateur complet.
- **Tests de régression** : Garantissent que les changements n'introduisent pas de nouvelles erreurs.