TP2 - Les bases des tests unitaires (TypeScript)

Objectifs

- Ecrire vos premiers tests unitaires avec **Vitest**
- Comprendre le pattern **Arrange / Act / Assert**
- Utiliser les assertions de base : `expect().toBe`, `expect().toBe(true)`, `expect(() => ...).toThrow()`
- Decouvrir les tests parametres avec `it.each`

Mise en place

Creez un nouveau projet TypeScript avec Vitest :

```
mkdir testing-tp2
cd testing-tp2
npm init -y
npm install -D vitest typescript @types/node
```

Ajoutez cette configuration dans `package.json` :

```
{
  "scripts": {
    "test": "vitest",
    "test:run": "vitest run"
  }
}
```

Creez un fichier `tsconfig.json` :

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "ESNext",
    "moduleResolution": "Bundler",
    "strict": true,
    "outDir": "dist"
  },
  "include": ["src/**/*", "tests/**/*"]
}
```

Creez les dossiers `src/` et `tests/`.

Exercice 1 - StringCalculator

Contexte

Un collegue a ecrit une classe `StringCalculator`. La methode `add` prend une chaine de caracteres contenant des nombres separes par des virgules et retourne leur somme.

Exemples attendus :

- `""` → `0`
- `"1"` → `1`
- `"1,2"` → `3`
- `"1,2,3,4,5"` → `15`

Voici son implementation dans `src/StringCalculator.ts` :

```
export class StringCalculator {
    add(numbers: string): number {
        if (!numbers || numbers.length === 0)
            return 0;

        const parts = numbers.split(',');

        let sum = 0;
        for (let i = 1; i <= parts.length; i++) {
```

```
            sum += parseInt(parts[i]);
        }

        return sum;
    }
}
```

Objectif

Votre mission est d'ecrire des tests unitaires pour valider le comportement de cette methode. Creez le fichier `tests/StringCalculator.test.ts` et completez-le :

```
import { describe, it, expect } from 'vitest';
import { StringCalculator } from '../src/StringCalculator';

describe('StringCalculator', () => {
    it('add - chaine vide retourne 0', () => {
        // Arrange
        const calculator = new StringCalculator();

        // Act
        const result = calculator.add('');

        // Assert
        expect(result).toBe(0);
    });

    // A vous d'ajouter d'autres tests !
});
```

Etape 1 - Tester les cas de base

Ecrivez un test pour chacun des cas suivants :

1. Une chaine vide `""` doit retourner `0`
2. Un seul nombre `"5"` doit retourner `5`
3. Deux nombres `"1,2"` doivent retourner `3`
4. Plusieurs nombres `"1,2,3,4,5"` doivent retourner `15`

Lancez vos tests avec `npm test`. Que constatez-vous ?

Etape 2 - Analyser les echecs

Certains de vos tests echouent. Pour chaque echec :

1. Lisez le message d'erreur
2. Identifiez la ligne du code source qui pose probleme
3. Expliquez le bug en une phrase

Etape 3 - Corriger le code

Corrigez l'implementation de `StringCalculator` pour que tous vos tests passent.

Etape 4 - Cas limites

Ecrivez des tests supplementaires pour ces cas limites :

Cas	Entree	Resultat attendu
Espaces autour des nombres	`"1, 2, 3"`	`6`
`undefined` en entree	`undefined as any`	`0`
Un seul nombre avec virgule	`"7,"`	`7`

Vos tests passent-ils ? Si non, adaptez l'implementation.

Etape 5 - Refactoring avec `it.each`

Vous remarquez que vos tests se ressemblent beaucoup. Refactorisez-les en un seul test parametre.

Vitest permet de parametrer les tests avec `it.each` :

```
it.each([
    ['entree1', resultatAttendu1],
    ['entree2', resultatAttendu2],
] as [string, number][])('add("%s") retourne %d', (input, expected) => {
    // ...
});
```

Refactorisez vos tests des etapes 1 a 4 en utilisant ce pattern. Choisissez quels cas regrouper et comment nommer le test.

Verifiez que chaque entree du tableau genere un test distinct dans le rapport.

Etape 6 - Nombres negatifs interdits

La calculatrice doit lever une erreur si l'entree contient un nombre negatif.

Ecrivez un test qui verifie ce comportement :

```
it('add - nombre negatif leve une erreur', () => {
    const calculator = new StringCalculator();

    expect(() => calculator.add('1,-2,3')).toThrow(Error);
    expect(() => calculator.add('1,-2,3')).toThrow('-2');
});
```

Implementez cette regle dans `StringCalculator` :

```
throw new Error(`Nombres negatifs interdits : ${negatives.join(', ')}`);
```

Etape 7 - Delimiteur personnalise

Ajoutez le support d'un delimiteur personnalise. Le format est : `"//[delimiteur]\n[nombres]"`

Exemples :

- `"//;\n1;2"` → `3`
- `"//|\n4|5|6"` → `15`

Ecrivez les tests d'abord, puis l'implementation.

Exercice 2 - PasswordValidator

Contexte

Vous allez creer un validateur de mot de passe et le tester rigoureusement. Le but est de pratiquer les tests parametres et la gestion de cas limites.

Regles de validation

Un mot de passe est valide si **toutes** les conditions suivantes sont remplies :

1. Au moins **8 caracteres**
2. Contient au moins **une lettre majuscule**
3. Contient au moins **une lettre minuscule**
4. Contient au moins **un chiffre**

Etape 1 - La classe de base

Creez la classe `PasswordValidator` dans `src/PasswordValidator.ts` :

```
export class PasswordValidator {
    isValid(password: string): boolean {
        throw new Error('Not implemented');
    }
}
```

Etape 2 - Tester les mots de passe valides

Commencez par verifier qu'un mot de passe respectant toutes les regles est accepte :

```
import { describe, it, expect } from 'vitest';
import { PasswordValidator } from '../src/PasswordValidator';

describe('PasswordValidator', () => {
    it.each([
        ['Abcdefg1'],
        ['MonMotDePasse9'],
        ['P4ssw0rdOk'],
    ] as [string][])('isValid("%s") retourne true', (password) => {
        const validator = new PasswordValidator();
```

```
        const result = validator.isValid(password);

        expect(result).toBe(true);
    });
});
```

Implementez `isValid` pour faire passer ces tests.

Etape 3 - Tester chaque regle individuellement

C'est a vous d'ecrire les tests. Pour chaque regle violee, verifiez que `isValid` retourne `false` :

Trop court (< 8 caracteres) :

```
it.each([
    ['Ab1'],
    ['Abcdef1'],
    // A completer...
] as [string][])('isValid - trop court: "%s" retourne false', (password) =
    // A vous !
});
```

Pas de majuscule :

```
it.each([
    ['abcdefg1'],
    // A completer...
] as [string][])('isValid - pas de majuscule: "%s" retourne false', (passw
    // A vous !
});
```

Pas de minuscule :

```
// A vous d'ecrire ce test !
```

Pas de chiffre :

```
// A vous d'ecrire ce test !
```

Etape 4 - Cas limites

Ecrivez des tests pour les cas suivants. Reflechissez au resultat attendu **avant** d'ecrire le test :

Cas	Mot de passe	Valide ?
Chaine vide	`""`	?
Exactement 8 caracteres	`"Abcdefg1"`	?
Que des chiffres	`"12345678"`	?
Que des majuscules + chiffre	`"ABCDEFG1"`	?
Mot de passe tres long	`"Aaaaaaaaaaaaaaa1"`	?

Etape 5 - Retourner des messages d'erreur

Refactorisez `PasswordValidator` pour retourner la **liste des regles violees** au lieu d'un simple booleen :

```
export interface ValidationResult {
    isValid: boolean;
    errors: string[];
}

export class PasswordValidator {
    validate(password: string): ValidationResult {
        // A implementer
    }
}
```

Ecrivez les tests correspondants. Par exemple :

```
it('validate - trop court et pas de chiffre retourne les deux erreurs', ()
    const validator = new PasswordValidator();

    const result = validator.validate('Abcde');

    expect(result.isValid).toBe(false);
    expect(result.errors).toHaveLength(2);
    expect(result.errors.some(e => e.includes('8 caracteres'))).toBe(true)
    expect(result.errors.some(e => e.includes('chiffre'))).toBe(true);
});
```

Etape 6 - A vous de jouer

Ajoutez une nouvelle regle : le mot de passe doit contenir au moins **un caractere special** parmi `!@#$%^&*`.

1. Ecrivez d'abord les tests
2. Verifiez qu'ils echouent
3. Implementez la regle
4. Verifiez que tous les tests passent

N'oubliez pas de mettre a jour vos tests existants pour qu'ils restent valides avec cette nouvelle regle !

Exercice 3 - ShoppingCart

Contexte

Vous allez tester un panier d'achat. Contrairement aux exercices precedents, `ShoppingCart` est un **objet a etat** : ses methodes modifient l'objet, et l'ordre des operations compte.

> **Note** : Vitest recree les variables declarees dans chaque test de facon isolee. Chaque test demarre avec un panier tout neuf — pas besoin de le reinitialiser manuellement.

Etape 1 - Tester un panier vide

Creez `src/ShoppingCart.ts` avec la signature suivante :

```
export class ShoppingCart {
    add(productName: string, unitPrice: number, quantity: number = 1): voi
        throw new Error('Not implemented');
    }
    remove(productName: string): boolean {
        throw new Error('Not implemented');
    }
    getTotal(): number {
        throw new Error('Not implemented');
    }
    get count(): number {
        throw new Error('Not implemented');
    }
}
```

Creez `tests/ShoppingCart.test.ts` et ecrivez le test suivant :

```
import { describe, it, expect } from 'vitest';
import { ShoppingCart } from '../src/ShoppingCart';

describe('ShoppingCart', () => {
    it('nouveau panier - count est 0 et total est 0', () => {
        const cart = new ShoppingCart();

        expect(cart.count).toBe(0);
        expect(cart.getTotal()).toBe(0);
    });
});
```

Ce test doit **echouer**. C'est votre point de depart.

Etape 2 - Tester l'ajout d'articles

Implementez `add` et `getTotal`, puis ecrivez les tests suivants :

1. Ajouter un article → `getTotal()` retourne `unitPrice * quantity`

2. Ajouter deux articles differents → `getTotal()` retourne la somme
3. `count` retourne le nombre de **produits distincts** (pas la somme des quantites)

```
it('add - un article - total egal au prix fois quantite', () => {
    const cart = new ShoppingCart();

    cart.add('Stylo', 1.50, 3);

    expect(cart.getTotal()).toBe(4.50);
    expect(cart.count).toBe(1);
});
```

Ecrivez les deux autres tests vous-meme.

Etape 3 - Tester la suppression

Implementez `remove`, puis ecrivez les tests :

1. Supprimer un article present → retourne `true`, le total est mis a jour
2. Supprimer un article absent → retourne `false`, le panier est inchange
3. Supprimer le dernier article → `count` revient a `0`

Etape 4 - TDD : Remise automatique

Ecrivez les tests avant l'implementation.

Nouvelle regle metier : si le total **avant remise** est superieur ou egal a `100`, une remise de 10 % est appliquee.

Exemples :

- Total brut `80` → `getTotal()` retourne `80` (pas de remise)
- Total brut `100` → `getTotal()` retourne `90`
- Total brut `150` → `getTotal()` retourne `135`

Cycle a suivre :

1. Ecrivez un test pour le cas "pas de remise" — il doit passer (comportement existant)
2. Ecrivez un test pour le cas "remise a 100" — il doit **echouer**

3. Implementez la remise
4. Verifiez que **tous** les tests passent (y compris ceux des etapes 1-3)

Recapitulatif

Concept C#	Equivalent TypeScript (Vitest)	Ou l'avez-vous pratique ?
`[Fact]`	`it()` / `test()`	Exercice 1, etapes 1 a 4
`[Theory]` + `[InlineData]`	`it.each()`	Exercice 1 etape 5, Exercice 2
`Assert.Equal(expected, actual)`	`expect(actual).toBe(expected)`	Exercices 1 et 3
`Assert.True(result)`	`expect(result).toBe(true)`	Exercices 2 et 3
`Assert.False(result)`	`expect(result).toBe(false)`	Exercices 2 et 3
`Assert.Throws<T>(() => fn())`	`expect(() => fn()).toThrow()`	Exercice 1 etape 6
`Assert.Contains(item, list)`	`expect(list).toContain(item)`	Exercice 2 etape 5
Pattern Arrange / Act / Assert	Pattern identique	Partout
Tester du code existant	Meme approche	Exercice 1, etapes 1 a 3

Tester les cas limites	Meme approche	Exercices 1, 2 et 3
Tester un objet a etat	Meme approche	Exercice 3
Isolation par defaut (instance par test)	Meme comportement	Exercice 3
Ecrire le test avant le code (TDD)	TDD	Exercice 1 etape 7, Exercice 2, Exercice 3 etape 4