

Qualité de développement

Tests unitaires

Principe de base

Un code non testé n'a aucune valeur.

Définition d'un test

Un test est un programme qui vérifie que le comportement d'un autre programme est conforme à ce qui est attendu. Si le comportement n'est pas conforme, le test échoue et indique le problème.

Définition d'un test unitaire

Un test unitaire est un test qui vérifie le comportement d'une seule unité de code. Une unité de code est une partie du programme qui est suffisamment petite pour être testée de manière autonome. Cas typique : on teste **une** fonction, **une** méthode de classe.

Un programme de tests va contenir une multitude de tests unitaires.

Remarque : il existe d'autres types de tests qui seront vus plus tard dans votre cursus.

Test unitaire en TypeScript

Il existe de nombreux outils pour effectuer des tests en TypeScript (et dans d'autres langages). Dans le cours, nous allons utiliser les outils inclus directement dans *deno*.

L'ensemble des tests à réaliser est défini par le développeur. Ces outils servent à automatiser leur exécution et à fournir une synthèse finale.

Il y a des différences de syntaxe entre les différents outils mais ils reposent sur l'utilisation d'**assertions**.

Assertions

Une assertion est une instruction qui vérifie qu'une condition est vraie. Si la condition est fausse, l'assertion échoue.

Il peut s'agir de vérifier que la valeur retournée par une fonction est bien celle attendue ou qu'une valeur se trouve dans un tableau.

Dans *deno*, il existe une vingtaine d'assertions dont vous trouverez la liste [ici](#).

Remarque : dans le cours, on peut avoir l'impression que les assertions ne peuvent tester que des retours de fonctions. En réalité, on peut tester n'importe quelle expression. On utilise souvent des fonctions car elles forment une bonne unité de programme pour les tests unitaires.

Voici un exemple d'utilisation de l'assertion `assertEquals` (égalité) dans un test unitaire :

```
import { assertEquals } from "jsr:@std/assert/equals";

Deno.test("test d'existence d'une URL", () => {
  const url = new URL("./foo.js", "https://deno.land/");
  assertEquals(url.href, "https://deno.land/foo.js");
});
```

Dans l'exemple, on construit une URL et ensuite on vérifie que l'URL construite est bien celle attendue.

On peut ensuite exécuter le test en utilisant la commande suivante (en supposant que le code précédent est dans un fichier `url.test.ts`) :

```
% deno test url.test.ts
Check file:///home/laroche5/ens/tu/tests/url.test.ts
running 1 test from ./url.test.ts
test d'existence d'une URL ... ok (0ms)

ok | 1 passed | 0 failed (1ms)
```

On constate que le test a réussi.

Syntaxe d'un test

Il faut commencer par importer les assertions que l'on souhaite utiliser :

```
import { assertEquals } from "jsr:@std/assert/equals";
```

La syntaxe d'un test est la suivante :

```
Deno.test("nom du test", () => {
  // instructions du test
});
```

Indication : `() => {}` est une fonction anonyme sans paramètre.

Le corps de la fonction anonyme contient les instructions du test. Elle permet de déclarer si besoin des variables, faire des calculs et appeler les assertions.

Mise en place d'une campagne de test

Bonne pratique : il est indispensable de séparer les tests du code à tester. En effet, cela permet de ne pas mélanger les deux et de pouvoir exécuter les tests sans avoir à exécuter le code à tester. Dans le cours on suppose que les tests sont dans un dossier `tests` et le code à tester dans un dossier `src`.

Supposons que l'on souhaite tester la fonction suivante stockée dans un fichier `src/MesMaths.ts` :

```
export function multAdd(x: number, y: number): number {
  let resultat = 0;

  for (let i = 0; i < y; i++) {
    resultat += x;
  }

  return resultat;
}
```

Rappel : le mot-clef `export` permet de rendre la fonction visible depuis l'extérieur du fichier.

Pour tester cette fonction, on va créer un fichier `tests/MesMaths.test.ts` qui contiendra les tests à réaliser.

Première étape : définir les **cas de test**.

- Le cas général : une multiplication avec des nombres sans aucune particularité (exemple : $3 \times 8 = 24$)
- Puis, on cherche les **tests aux limites**. Les limites sont liées au domaine étudié (ici, les maths), et, souvent, aux types de données gérées (ici, des entiers... ou des réels ???)
 - Lié au domaine : "0 est absorbant pour la multiplication". Le cas "0" est un cas particulier, je dois le tester
 - Lié au domaine : gestion des négatifs. Que se passe-t-il si je multiplie -2 ? Que se passe-t-il si je multiplie par -3 ?
 - Lié aux types de données : la fonction a des paramètres de type `number`, donc elle accepte les réels. Fonctionne-t-elle si je multiplie 2.5 ? Si je multiplie par 2.5 ?
 - Lié au langage / aux détails d'implémentation : que se passe-t-il si j'utilise une des valeurs suivantes ?

```
Number.NaN
Number.MAX_VALUE
```

```
Number.MAX_SAFE_INTEGER
Number.POSITIVE_INFINITY
```

- Suite à cette réflexion, on peut programmer les tests (ce qui suit n'est qu'un extrait du fichier)

```
import { multAdd } from "../src/MesMaths.ts";
import { assertEquals } from "jsr:@std/assert/equals";
import { assertAlmostEquals } from "jsr:@std/assert/almost-equals";

Deno.test("cas général", () => {
  assertEquals(multAdd(3, 8), 24);
});

Deno.test("multiplication par 0", () => {
  assertEquals(multAdd(3, 0), 0);
});

Deno.test("multiplication par un nombre négatif", () => {
  assertEquals(multAdd(3, -8), -24);
});
```

On peut alors exécuter automatiquement les tests définis et on obtient alors le résultat suivant :

```
% deno test tests/MesMaths.test.ts
Check file:///home/laroche5/ens/tu/tests/MesMaths.test.ts
running 3 tests from ./tests/MesMaths.test.ts
cas général ... ok (5ms)
multiplication par 0 ... ok (3ms)
multiplication par un nombre négatif ... FAILED (5ms)

ERRORS

multiplication par un nombre négatif => ./tests/MesMaths.test.ts:12:6
error: AssertionError: Values are not equal.

[Diff] Actual / Expected

- 0
+ -24

throw new AssertionError(message);
    ^
    at assertEquals (https://deno.land/std@0.220.0/assert/assert_equals.ts:53:9)
```

```
at file:///home/laroche5/ens/tu/tests/MesMaths.test.ts:13:3
```

FAILURES

multiplication par un nombre négatif => ./tests/MesMaths.test.ts:12:6

FAILED | 2 passed | 1 failed (37ms)

error: Test failed

On constate que le test **multiplication par un nombre négatif** a échoué. En effet, la fonction **multAdd** ne gère pas le cas où le nombre de fois où on multiplie est négatif et retourne alors 0 au lieu de -24. Les deux autres tests ont réussi.

Refactoring suite au test : Pour faire réussir tous mes tests, je vais modifier la fonction **multAdd**.

Sans un framework de test :

- je modifie mon code pour prendre en compte le cas "multiplication par un nombre négatif" ;
- je reteste avec -5 ou autre nombre négatif ;
- je m'arrête de travailler lorsque ce cas fonctionne.

Mais mes modifications ont peut-être introduit d'autres erreurs, sur des cas qui fonctionnaient auparavant. On appelle cela une **régression**.

Avec un framework de test :

- je modifie mon code pour prendre en compte le cas "multiplication par un nombre négatif" ;
- je relance **toute ma batterie de tests** ;
- je m'arrête lorsque tous les tests réussissent.

Test des méthodes d'un objet

On considère la classe suivante dans le fichier **src/Point.ts** :

```
export class Point {
  private _x: number;
  private _y: number;

  constructor(x = 0, y = 0) {
    this._x = x;
    this._y = y;
  }

  get x(): number {
    return this._x;
  }
}
```

```

get y(): number {
    return this._y;
}

déplacer(dx: number, dy: number): void {
    this._x += dx;
    this._y += dy;
}

distance(p: Point): number {
    const dx = this._x - p._x;
    const dy = this._y - p._y;
    return Math.sqrt(dx * dx + dy * dy);
}

toString(): string {
    return "(" + this._x + this._y + ")";
}

copier(): Point {
    return new Point(this._x, this._y);
}

```

Un fichier de test `tests/Point.test.ts`, qui teste unitairement la classe `Point`, peut être le suivant :

```

import { Point } from "../src/Point.ts";
import { assertEquals } from "jsr:@std/assert/equals";

Deno.test("constructeur, cas par défaut", () => {
    const p = new Point();
    assertEquals(p.x, 0);
    assertEquals(p.y, 0);
});

Deno.test("constructeur, cas avec paramètres", () => {
    const p = new Point(3, 5);
    assertEquals(p.x, 3);
    assertEquals(p.y, 5);
});

Deno.test("déplacement", () => {
    const p = new Point(3, 5);
    p.déplacer(2, 3);
    assertEquals(p.x, 5);
    assertEquals(p.y, 8);
});

```

```

Deno.test("distance", () => {
  const p1 = new Point(3, 5);
  const p2 = new Point(5, 8);
  assertAlmostEquals(p1.distance(p2), 3.605551275);
});

Deno.test("toString", () => {
  const p = new Point(3, 5);
  assertEquals(p.toString(), "(3,5)");
});

Deno.test("copier", () => {
  const p1 = new Point(3, 5);
  const p2 = p1.copier();
  assertEquals(p1.x, p2.x);
  assertEquals(p1.y, p2.y);
});

```

Remarques :

- Chaque test est précédé d'un texte indiquant clairement ce qu'on teste (méthode et cas) : cela permettra de voir tout de suite le cas de test qui n'a pas fonctionné. Exemple :
`Deno.test("constructeur, cas par défaut", ...`
- Chaque méthode de la classe est testée au moins une fois
- Pour chaque méthode, on étudie les différents cas possibles et chaque cas donne lieu à un test unitaire.

L'exécution des tests va donner ce qui suit :

```

% deno test tests/point.test.ts
Check file:///home/laroche5/ens/tu/tests/Point.test.ts
running 6 tests from ./tests/Point.test.ts
constructeur par défaut ... ok (0ms)
constructeur avec paramètres ... ok (0ms)
déplacement ... ok (0ms)
distance ... ok (0ms)
toString ... FAILED (1ms)
copier ... ok (0ms)

```

ERRORS

```

toString => ./tests/Point.test.ts:29:6
error: AssertionError: Values are not equal.

```

[Diff] Actual / Expected

```
- (35)
+ (3,5)

throw new AssertionError(message);
    ^
    at assertEquals (https://deno.land/std@0.220.0/assert/assert_equals.ts:53:9)
    at file:///home/laroche5/ens/tu/tests/Point.test.ts:31:3

FAILURES

toString => ./tests/Point.test.ts:29:6

FAILED | 5 passed | 1 failed (3ms)

error: Test failed
```

On se rend compte que l'on a fait une erreur sur la méthode `toString` qui ne retourne pas la bonne chaîne de caractères. Il faudra donc corriger cette erreur.

Important : tous les tests doivent passer à la fin. Il s'agit de tester si le code a le comportement attendu. Si un test échoue, il faut corriger le code pour que le test passe. Néanmoins, si un test échoue :

- c'est que le code à tester n'est pas correct
- ou que le code de test est incorrect !!!

Test des classes d'une application

Dans une application, on écrit plusieurs classes. On aura alors un fichier de test par classe. Pour exécuter tous les fichiers de tests en une seule fois, on peut faire simplement : `deno test`. Tous les fichiers nommés avec une extension `.test.ts` trouvés dans les sous-répertoires du dossier courant seront alors exécutés.

Conclusions

- Les outils de tests permettent d'automatiser les tests d'une application ;
- cela permet de relancer en une seule commande tous les tests à chaque modification du code, et de vérifier que les modifications n'ont pas introduit de bugs (principe de **non-régression**) ;
- la définition des tests à réaliser est la tâche du développeur et ils doivent être pertinents et de qualité ;
 - un test doit être simple à comprendre et à écrire ;
 - un test unitaire, comme son nom l'indique, ne teste qu'une seule unité de programmation (1 fichier de test ne teste qu'une seule classe ; 1 test ne teste qu'un seul cas de test d'une seule méthode) ;

- un test doit être reproductible : il doit toujours donner le même résultat quelque soit le contexte.