

Développement front

MMI 3 – TP#1 S5

Danielo **JEAN-LOUIS**

But du cours

- Approfondir les connaissances de nodejs
- Découverte des tests unitaires et e2e tests*
- Préparation à la SAE 501

* : end-to-end test ou tests de bout en bout

Test unitaire

- Appelé également U.T. / T.U.
 - Et Unit Test en anglais
- Inventé au milieu des années 90
- Pierre angulaire des méthodes Extreme Programming (XP) et Test Driven Development (TDD)
- Existent dans quasiment tous les langages

Source(s) :

- <https://www.artofunittesting.com/definition-of-a-unit-test>

Test unitaire

- Permet de tester *toutes* les possibilités d'une unité de code
 - Est-ce que ma fonction retourne la bonne valeur ?
 - Comment se comporte ma fonction si une valeur est absente/incorrecte ?
 - ...
- Encourage / séparation du propreté du code

Source(s) :

- <https://www.artofunittesting.com/definition-of-a-unit-test>

Point technique – Unité de code

- Petite portion de code : fonction, méthode...
- Plus "petit" sera votre code, plus facile sera sa relecture, son interprétation par les tests unitaires

Point technique – Unité de code

Autrement dit : une fonctionnalité = une fonction / méthode (si possible)

Test unitaire

- Limite la régression de code (introduction de - nouveaux - bugs)
- Facilite le travail de la QA (Assurance Qualité)
- Peut fonctionner de pair avec les critères d'acceptation
 - Un test validant un critère d'acceptation

Source(s) :

- <https://www.artofunittesting.com/definition-of-a-unit-test>

Test unitaire

- Retourne un booléen en fonction de l'exécution du test
 - True → ça s'est bien passé
 - False → ça s'est mal passé
- Sert de documentation "vivante"
- Doit être rejouable avec le même résultat

Source(s) :

- <https://www.artofunittesting.com/definition-of-a-unit-test>

Test unitaire

- Limite les erreurs humaines, les oublis de tests manuels...
- Écrit par les développeurs et exécutés par la machine
 - Avant un commit, avant un git merge...
- Privilégiez les fonctions dites “pures”

Source(s) :

- <https://www.artofunittesting.com/definition-of-a-unit-test>

Point technique – Fonction pure

- Concept transverse aux langages de programmation
- Fonction qui retourne **toujours** la même chose avec les mêmes paramètres
- Modifie les variables au sein de son propre bloc au lieu de variables globales
- Rend plus simple les tests unitaires

Point technique – Fonction pure



JS

```
const capitalize = (str) => {  
  return `${str.charAt(0).toUpperCase()}${str.slice(1)}`;  
}
```

Cette fonction est pure :

- Elle ne modifie / utilise aucune variable externe à la fonction
- Retourne toujours le même résultat avec le même argument

Test unitaire

"Un test unitaire est un code automatisé impliquant une petite unité de code et teste si elle produit le comportement attendu"

Source(s) :

- <https://www.artofunittesting.com/definition-of-a-unit-test>

Vitest

- Framework javascript pour les tests unitaires
- Outil open source et extensible
- Fonctionne avec l'outil Vite
 - Abordé plus tard
- Nécessite nodejs
 - Exécuté côté ordinateur, pas navigateur

Source(s) :

- <https://vitest.dev/>

Vitest

- Fonctionne avec n'importe quelle typologie d'environnement javascript : vanilla js, frameworks, nodejs...
- Propose un ensemble de méthodes pour tester son code

Source(s) :

- <https://vitest.dev/>

Vitest

- Propose une documentation complète et traduite en plusieurs langues (dont le français partiellement)
- Affiche les résultats dans le terminal
 - Et le navigateur (en option)
- **Outil de développement**
 - S'installe avec `npm install --save-dev vitest`

Source(s) :

- <https://vitest.dev/>

Vitest - Exemple



JS

```
const sum = (a, b) => {  
  return a + b;  
}
```

```
export sum;
```

← Définition de la fonction javascript

Définition du test associé →



JS

```
import { expect, test } from 'vitest';  
import { sum } from './sum';
```

```
test('adds 1 + 2 to equal 3', () => {  
  expect(sum(1, 2)).toBe(3);  
});
```

Source(s) :

- <https://vitest.dev/>

Vitest - Exemple



JS

```
import { expect, test } from 'vitest';  
import { sum } from './sum';
```

```
test('adds 1 + 2 to equal 3', () => {  
  expect(sum(1, 2)).toBe(3);  
});
```

Import de la fonction à tester

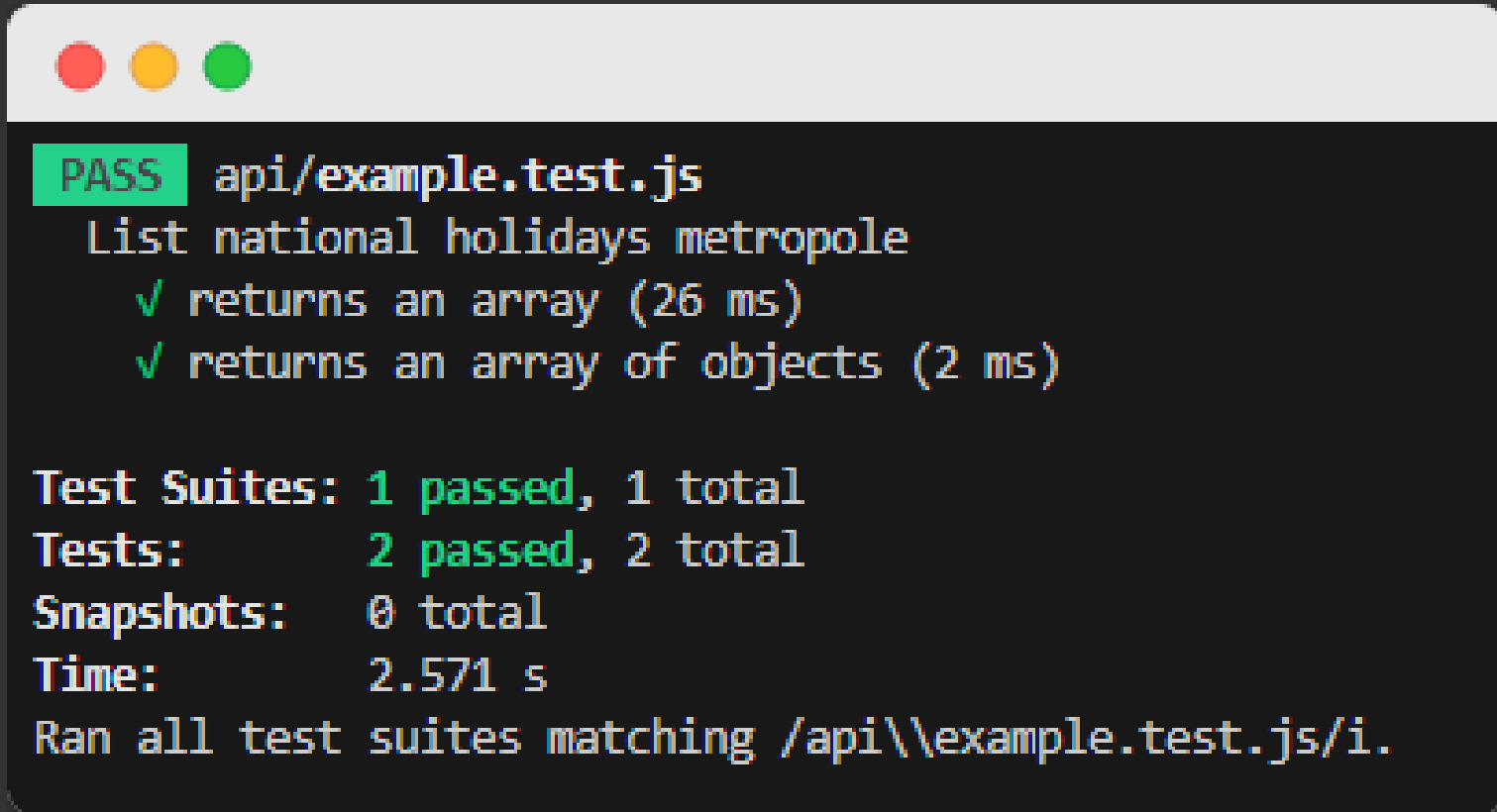
Conteneur du test (nom + fonction)

Définition du test. Dans notre cas, on cherche à savoir si le résultat de la fonction “sum()” avec les paramètres 1 et 2 est bien égal à 3

Source(s) :

- <https://vitest.dev/>

Vitest - Exemple



```
PASS api/example.test.js
  List national holidays metropole
    ✓ returns an array (26 ms)
    ✓ returns an array of objects (2 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        2.571 s
Ran all test suites matching /api\\example.test.js/i.
```

Tous nos tests se sont bien déroulés

Source(s) :

- <https://vitest.dev/>

Le résultat des tests peut être affiché dans le terminal ou dans le navigateur

Vitest - Exemple

- Le nom d'un test doit être court et clair
- La méthode “test()” peut être remplacée par “it()”

Source(s) :

- <https://vitest.dev/>

**La même fonction / méthode peut être testée
plusieurs fois avec des paramètres différents,
configurations différentes**

Vitest - expect()

- Méthode native de vitest
- Permet d'exécuter une fonction
- A utiliser quand vous devez valider un comportement
- Toujours suivi par un test d'égalité comme la méthode toBe()

Source(s) :

- <https://jestjs.io/docs/expect#expectvalue>

Vitest - toBe()

- Méthode native de vitest
- Un des nombreux comparateur (matcher) de vitest
- Équivalent à “égal à” en javascript
 - Équivalence stricte. Test de la valeur et du type

Source(s) :

- <https://jestjs.io/docs/expect#tobevalue>

Pratiquons ! - Découvrons vitest (Partie 1/2)

Pré-requis :

- Avoir la ressource ressources/vitest/initiation

A télécharger ici :

https://github.com/DanYellow/cours/raw/refs/heads/main/s5-developpement-front/travaux-pratiques/numero-1/s5-developpement-front_travaux-pratiques_numero-1.ressources.zip

Conventions

- Extension de fichiers .spec.js ou .test.js
 - A coté du fichier “normal” ou un dossier “tests”
- Nommages identiques (avant l’extension) du fichier javascript et de test
 - Ex : foo.js → foo.test.js ou foo.spec.js
- **Le nom d’un test doit être explicite**
- Grouper les tests par catégorie / fonctionnalité

Conventions

- Commencer le nom d'un test par "should"
- Mettre dans le nom du test le résultat attendu
 - Ex : (it) should increase balance


Grouper les tests

- Permet de structurer ses tests en fonction du thème
- Utilisation de la méthode “describe()”
 - Possibilité d’imbriquer les groupes de tests

Source(s) :

- <https://jestjs.io/docs/api#describename-fn>

Grouper les tests

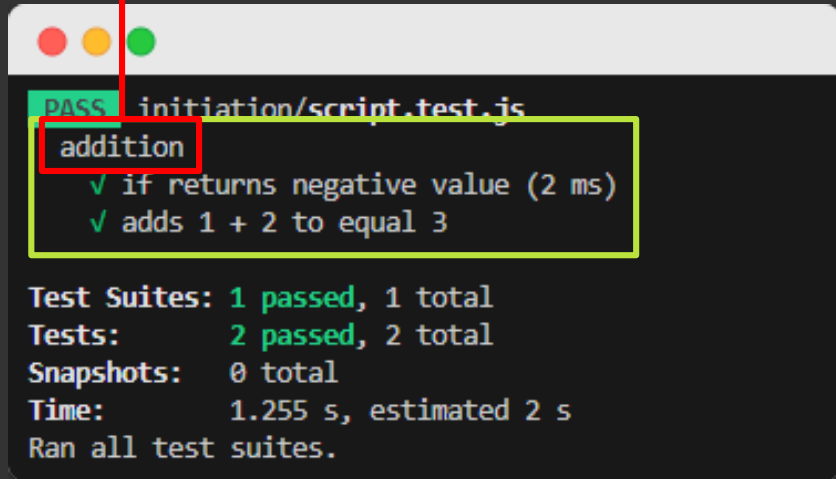


```
describe("addition", () => {  
  test("if returns negative value", () => {  
    expect(sum(1, -2)).toBe(-1);  
  });  
  
  test("adds 1 + 2 to equal 3", () => {  
    expect(sum(1, 2)).toBe(3);  
  });  
});
```

On groupe tous nos tests liés à l'addition

Source(s) :

- <https://jestjs.io/docs/api#describename-fn>



```
initiation/script.test.js  
PASS  
  addition  
    ✓ if returns negative value (2 ms)  
    ✓ adds 1 + 2 to equal 3  
  
Test Suites: 1 passed, 1 total  
Tests:      2 passed, 2 total  
Snapshots:  0 total  
Time:       1.255 s, estimated 2 s  
Ran all test suites.
```

Dans la console, les tests sont groupés sous le nom "addition"

Vites – Tests possibles (liste non exhaustive)

- Longueur d'un tableau (.toHaveLength(valeur))
- Présence d'une clef dans un dictionnaire
 - (.toHaveProperty('clé'))
- Nullité d'une valeur (.toBeNull())
- Appel d'une fonction (.toBeCalled())
- Supérieur ou égal à (.toBeGreaterThanOrEqual(valeur))
- ...

Source(s) :

- <https://vitest.dev/api/expect.html>

Conseils

- Vous n'avez pas à tester tous les cas du monde
 - Pensez simple et réaliste
- Omettez les tests de cas quasi-impossibles
- Faites un plan de tests

Vitest - Modifiers

- Permettent de modifier le comportement d'une méthode (`expect()`, `test()`...)
- Exemple :
 - Exclure un test → `it.skip("[...]", () => {})`
 - Exclure un groupe de tests → `describe.skip()`
 - Tester le résultat inverse → `expect().not.toHaveLength(5)`

Source(s) :

- <https://jestjs.io/docs/expect#modifiers>

Tests asynchrones

- Tests effectués impliquant des tâches asynchrones → Promesses / Appel d'API
- Doivent être substitués par un mock pour en contrôler le comportement

mock

- Également appelé “spy”
- Permet de simuler une fonction existante
- Permet de changer le comportement en fonction des paramètres définis dans le test
- Désigne également de fausses données

mock

- Utile quand :
 - Une fonction effectue un comportement complexe
 - Une fonction retourne une donnée que vous ne contrôlez pas
 - Exemple une API
 - Vous souhaitez savoir si une fonction a été appelée

mock

- Doit avoir le même comportement que le code non-factice
 - Du moins doit retourner le même type de valeur que la fonction originale
- Copie contrôlée du vrai code
- Vous pouvez mocker la même fonction au global et au sein d'un test

mock



```
import { vi } from "vitest";

vi.mock("./file-to-mock.js", () => ({
  namedExport: vi.fn(param) => {
    // [mocked function]
  }),
  default: vi.fn(() => {
    // [mocked function]
  })
}));
```

On récupère le contenu du fichier "file-to-mock.js" et on mock les fonctions nommées et la fonction par défaut

mock



```
import { vi } from "vitest";  
import { namedExport } from "../file-to-mock.js";  
  
it("should call namedExport", () => {  
  // [...]  
  expect(namedExport).toBeCalled();  
});
```

On teste si notre fonction **mockée** a été appelée (par une autre fonction)

Point technique – Tests unitaires et API

- Ce n'est pas à vous (dev front) de tester une API. **Vous ne devez tester que ce que vous en faites**
- Si votre outil de test permet l'appel d'API. Préférez le mock quand même
 - Pas d'attente, possibilité de modifier la réponse...

Pratiquons ! - Test asynchrone (Partie 1)

Pré-requis :

- Avoir la ressource `ressources/vitest/api`

A télécharger ici :

https://github.com/DanYellow/cours/raw/refs/heads/main/s5-developpement-front/travaux-pratiques/numero-1/s5-developpement-front_travaux-pratiques_numero-1.ressources.zip

Mock – Attraper les erreurs

- Utilisation d'un try/catch/(finally) dans le test unitaire
- Éviter le crash des tests si jamais une erreur est levée

Source(s) :

- <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Statements/try...catch>

Mock – Attraper les erreurs



```
vi.mock("./file-to-mock.js", () => ({  
  namedExport: vi.fn().mockRejectedValue(new Error('Async error'));  
}));
```

↑ Notre mock lève une erreur grâce à la méthode "reject()"

Le test associé utilise un try/catch pour capturer l'erreur →



```
it("throw an error", () => {  
  try {  
    await namedExport();  
  } catch (error) {  
    await expect(namedExport).rejects.toThrowError();  
  }  
});
```

Source(s) :

- <https://vitest.dev/api/expect.html#tothrowerror>
- <https://vitest.dev/api/mock.html#mockrejectedvalue>

Mock API

- Proxifiez vos appels d'API pour mieux les manipuler dans vos tests
 - Voir exemple fourni dans la ressource
- Si vous avez beaucoup de données simulées créez un dossier dédié à vos mocks avec des fichiers associés

Pratiquons ! - Test asynchrone (Partie 2)

Pré-requis :

- Avoir la ressource ressources/jest/api

A télécharger ici :

https://github.com/DanYellow/cours/raw/refs/heads/main/s5-developpement-front/travaux-pratiques/numero-1/s5-developpement-front_travaux-pratiques_numero-1.ressources.zip

Limites

- Nécessite une maintenance du développeur
 - Màj du code = màj des tests unitaires
- Permet de tester que les fonctions
 - Le test d'interface ne doit pas être remplacé par les tests unitaires

Limites

- Peut prendre beaucoup, beaucoup de temps à exécuter
 - Nécessite parfois un ordinateur puissant
- Ne fonctionne pas avec l'entièreté d'un projet
 - S'arrête là où les tests d'intégration commencent

Nébuleuse de tests automatisés

- Tests unitaires
 - Testent une unité de code
- Tests d'intégration
 - Testent le fonctionnement orchestré de portions de code → ex : un formulaire entier
- Tests end-to-end
 - Simulent les actions utilisateurs. Testent un parcours entier

Cycle de vie d'un test

1. Initialisation (setUp) :

- Définition de l'environnement

2. Exécution

- Test du code

3. Vérification

- L'outil retourne vrai ou faux en fonction des assertions

4. Désactivation (tearDown)

- Nettoyage de l'environnement

Source(s) :

- <https://jestjs.io/docs/setup-teardown>

Cycle de vie d'un test

- Les phases d'initialisation (setUp) et de désactivation (tearDown) servent à limiter les effets de bord, réinitialiser les états

Source(s) :

- <https://jestjs.io/docs/setup-teardown>

Comment tester ses tests - coverage

- Appelé “couverture de code” en français
- Représente le pourcentage de code couvert par les tests unitaires
 - Quelle partie du code a été testée ?

Source(s) :

- <https://jestjs.io/blog/2020/01/21/jest-25#v8-code-coverage>

Comment tester ses tests - coverage

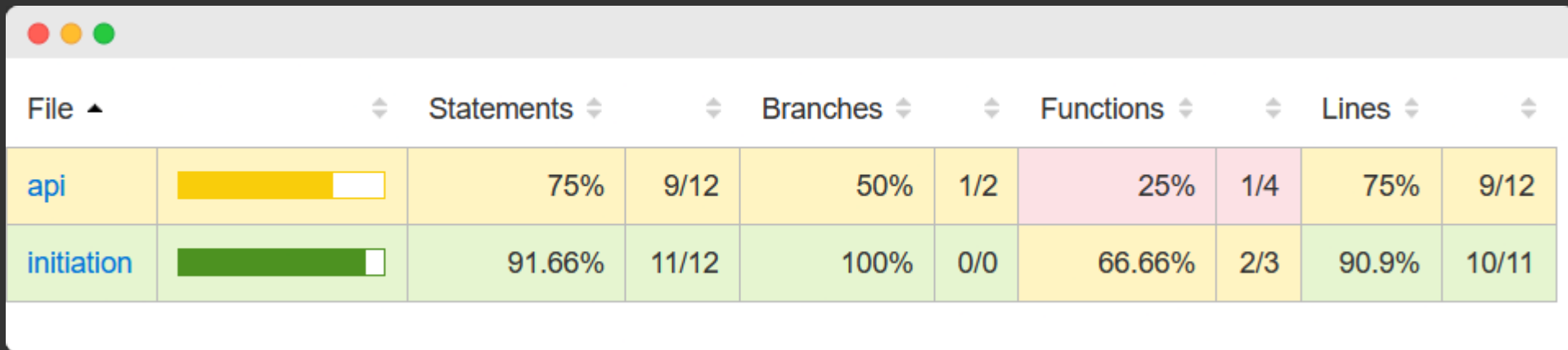
- Plus les chiffres sont hauts, plus le code est couvert mais...
 - ...ce n'est pas une donnée représentative de la qualité du projet
 - **100 % coverage != 0 % bugs**



Source(s) :

- <https://jestjs.io/blog/2020/01/21/jest-25#v8-code-coverage>

Code coverage

- Géré par l'outil Istanbul ou v8
- Génère un mini-site (dossier coverage/) détaillant la couverture du code ligne par ligne



File ▲		Statements ▾		Branches ▾		Functions ▾		Lines ▾	
api		75%	9/12	50%	1/2	25%	1/4	75%	9/12
initiation		91.66%	11/12	100%	0/0	66.66%	2/3	90.9%	10/11

Istanbul / v8 génère un résumé de la couverture du code.
Chaque fichier peut être ouvert pour avoir plus de détails

Code coverage

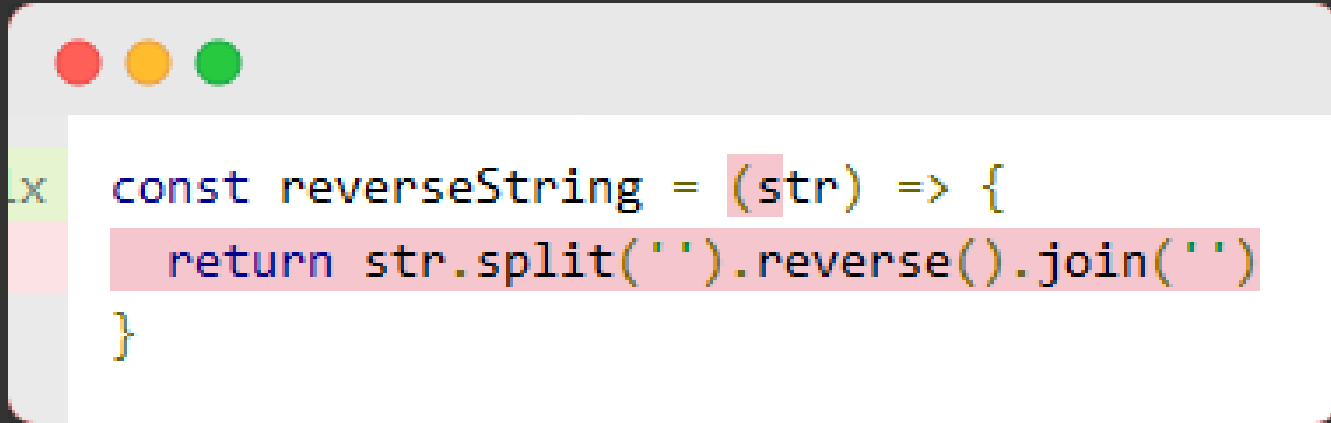
- Statements
 - Combien de déclarations ont été appelées (variables, return...) ?
- Branches
 - Quel pourcentage de conditions (if/else) ont été appelées ?

Code coverage

- Functions
 - Combien de fonctions ont été appelées ?
- Lines
 - Combien de lignes ont été exécutées ?

Code coverage

- Calcul du coverage
 - Code exécuté / code non-exécuté



```
x  const reverseString = (str) => {  
    return str.split('').reverse().join('')  
  }
```

Istanbul indique que la fonction n'a pas été appelée lors des tests

Code coverage

- Ne commitez pas les résultats. Le dossier coverage/ doit être dans le fichier .gitignore

Pratiquons ! - Test asynchrone (Partie 3)

Pré-requis :

- Avoir la ressource `ressources/vitest/api`

A télécharger ici :

https://github.com/DanYellow/cours/raw/refs/heads/main/s5-developpement-front/travaux-pratiques/numero-1/s5-developpement-front_travaux-pratiques_numero-1.ressources.zip

Code coverage - configuration

- Certaines parties / fichiers n'ont pas à être testés. Ils peuvent être ignorés
 - Configurable dans le fichier `vitest.config.js`

vitest.config.js

- Facultatif
- Permet de configurer le comportement de vitest sur plusieurs points :
 - Fichier à tester
 - Fichiers d'environnement à utiliser
 - ...

Source(s) :

- <https://vitest.dev/config/>

Questions ?

