**Jest Documentation**

1. **Definition**

Jest est un framework de test JavaScript développé par Facebook. Il est principalement utilisé pour tester des applications et des bibliothèques écrites avec React, mais il peut également être utilisé pour d'autres frameworks JavaScript tels que Angular et Vue.js.

1. **Premiers pas**

**Une image contenant texte, Police, capture d’écran, blanc

Description générée automatiquement**

"scripts": {

    "test":"jest \_\_tests\_\_ --coverage"

  },

**npm run test == >Will run all test in \_\_tests\_\_ folder et nous donne un coverage**

1. **Describe & Test**

En Jest les fonctions **`describe`** et **`test`** sont utilizer pour structurer et écrire des suites de tests.

La fonction "test" de Jest est utilisée pour définir des cas de test unitaires et vérifier le comportement attendu d'un code. "Describe" est utilisé pour regrouper des cas de test liés et les organiser de manière cohérente.

1. **Utilisation des comparateurs**

Jest propose plusieurs comparateurs couramment utilisés pour effectuer des assertions dans les tests. Voici quelques-uns des comparateurs les plus utilisés en Jest :

1. **expect().toBe(value)** : Vérifie si la valeur attendue est strictement égale (**===**) à la valeur reçue.
2. **expect().toEqual(value)** : Vérifie si la valeur attendue est équivalente à la valeur reçue en utilisant une comparaison récursive des propriétés pour les objets et les tableaux.
3. **expect().toBeNull()** : Vérifie si la valeur reçue est **null**.
4. **expect().toBeDefined()** : Vérifie si la valeur reçue est définie (**!== undefined**).
5. **expect().toBeUndefined()** : Vérifie si la valeur reçue est indéfinie (**=== undefined**).
6. **expect().toBeTruthy()** : Vérifie si la valeur reçue est évaluée à **true**.
7. **expect().toBeFalsy()** : Vérifie si la valeur reçue est évaluée à **false**.
8. **expect().toContain(item)** : Vérifie si un tableau ou une chaîne de caractères contient un élément spécifique.
9. **expect().toHaveLength(length)** : Vérifie si un tableau ou une chaîne de caractères a une longueur spécifique.
10. **expect().toHaveProperty(propertyPath, value?)** : Vérifie si un objet a une propriété spécifique avec une valeur facultative.
11. **expect().toMatch(regexpOrString)** : Vérifie si une chaîne de caractères correspond à une expression régulière ou à une sous-chaîne.
12. **expect().toThrow(error?)** : Vérifie si une fonction lance une exception, avec une option pour vérifier le type ou le message de l'exception.
13. **expect().not.toBe()**Vous pouvez également tester pour le contraire d'un comparateur en utilisant not

Ces comparateurs vous permettent de formuler des assertions précises dans vos tests pour vérifier le comportement attendu de votre code. Jest offre une large gamme de comparateurs pour couvrir différents cas de test et garantir la fiabilité de vos tests.

# **Tester du code asynchrone**

Il est fréquent en JavaScript que le code s'exécute de manière asynchrone. Lorsque vous avez du code qui s’exécute de façon asynchrone, Jest doit savoir quand le code qu'il teste est terminé, avant de passer à un autre test. Jest a plusieurs façons de gérer cela.

Pour tester du code asynchrone en utilisant Jest ,vous pouvez utilisez les fonctionaliter de gestion des promesses ou les fonctions asynchrone (`**async/await**`) , ou **done()** quand on utilise une fonction de rappel asynchrone .

* 1. Utiliser les promesses et les méthodes `**.then()`** et `**.catch()`**

Utilisation de la fonction then : Si votre code retourne une promesse, vous pouvez utiliser la fonction then pour définir les assertions à exécuter lorsque la promesse est résolue.

// Test avec then

test('Test asynchrone avec then', () => {

  return fetchData().then((result) => {

    expect(result).toBe('Données récupérées');

  });

});

* 1. Utilisation de la fonction **async/await** :

Utilisation de la fonction **async/await** : Vous pouvez marquer votre fonction de test comme **async** et utiliser le mot-clé **await** pour attendre la résolution de la promesse ou l'exécution d'une fonction asynchrone.

// Test avec async/await

test('Test asynchrone avec async/await', async () => {

  const result = await fetchData();

  expect(result).toBe('Données récupérées');

});

* 1. Utilisation de la fonction **expect** avec **resolves** :

Utilisation de la fonction expect avec resolves : Vous pouvez utiliser la fonction expect de Jest avec le modificateur resolves pour vérifier que la promesse est résolue avec une valeur spécifique.

// Test avec expect(resolves)

test('Test asynchrone avec expect(resolves)', () => {

  return expect(fetchData()).resolves.toBe('Données récupérées');

});

* 1. Utilisation de **done()**

Lorsqu'on utilise **done** dans Jest, on indique à Jest que le test asynchrone est terminé et qu'il peut continuer à exécuter les autres tests. **done** est généralement passé en tant que paramètre à une fonction de rappel asynchrone, et Jest attendra que **done** soit appelé pour marquer le test comme terminé.

// Fonction asynchrone à tester

function fetchData(callback) {

  // Supposons que fetchData effectue une requête asynchrone vers une API

  // et appelle le callback avec les données récupérées

  setTimeout(() => {

    const data = ['Donnée 1', 'Donnée 2', 'Donnée 3'];

    callback(data);

  }, 1000);

}

// Test avec done

test('Test asynchrone avec done', (done) => {

  fetchData((result) => {

    expect(result).toContain('Donnée 2');

    done();

// Appel de done() pour indiquer la fin du test asynchrone

  });

});

# **Mock Function ou Fonction simulées**

Les mock functions sont des fonctions simulées qui remplacent les fonctions réelles lors des tests. Elles permettent de contrôler le comportement des fonctions simulées et de vérifier leur utilisation dans les tests. Les mock functions peuvent enregistrer les appels qui leur sont faits, définir des valeurs de retour simulées et effectuer d'autres actions spécifiques pour les tests.

1. **Jest.fn()**

Elle permet de créer une fonction simulée (mock function) qui peut être utilisée pour simuler une fonction réelle lors des tests

describe("Utilisation d'une fonction simple", () => {

    const forEach = (items, callback) => {

      for (let index = 0; index < items.length; index++) {

        callback(items[index]);

      }

    };

    const mockCallback = jest.fn((x) => 42 + x);

    test("forEach mock function", () => {

      forEach([0, 1], mockCallback);

      // The mock function was called twice

      expect(mockCallback.mock.calls).toHaveLength(2);

      // The first argument of the first call to the function was 0

      expect(mockCallback.mock.calls[0][0]).toBe(0);

      // The first argument of the second call to the function was 1

      expect(mockCallback.mock.calls[1][0]).toBe(1);

      // The return value of the first call to the function was 42

      expect(mockCallback.mock.results[0].value).toBe(42);

    });

  });

1. **Propriete .mock**

 describe("Propriété .mock", () => {

    const myMock1 = jest.fn();

    const a = new myMock1();

    a.name = "a";

    console.log("a", a);

    const b = {};

    b.name = "b";

    const bound = myMock1.bind(b);

    bound();

    console.log("bound", bound());

    console.log("Instance Mock", myMock1.mock.instances);

    test("Some mock function", () => {

      const someMockFunction = jest.fn(() => "return value");

      someMockFunction("first arg", "second arg");

      // The function was called exactly once

      expect(someMockFunction.mock.calls).toHaveLength(1);

      // The first arg of the first call to the function was 'first arg'

      expect(someMockFunction.mock.calls[0][0]).toBe("first arg");

      // The second arg of the first call to the function was 'second arg'

      expect(someMockFunction.mock.calls[0][1]).toBe("second arg");

      // The return value of the first call to the function was 'return value'

      expect(someMockFunction.mock.results[0].value).toBe("return value");

      // The function was called with a certain `this` context: the `element` object.

      //   expect(someMockFunction.mock.contexts[0]).toBe(element);

      const someMockConstructor = jest.fn();

      const a = new someMockConstructor();

      const b = new someMockConstructor("test");

      a.name = "test";

      // This function was instantiated exactly twice

      expect(someMockConstructor.mock.instances.length).toBe(2);

      // The object returned by the first instantiation of this function

      // had a `name` property whose value was set to 'test'

      expect(someMockConstructor.mock.instances[0].name).toBe("test");

      // The first argument of the last call to the function was 'test'

      console.log("Test", someMockConstructor.mock.lastCall[0]);

      expect(someMockConstructor.mock.lastCall[0]).toBe("test");

    });

  });

1. **Valeur de retour simulées**

 describe("Valeurs de retour simulées", () => {

    const myMock = jest.fn();

    console.log(myMock());

    // > undefined

    myMock

      .mockReturnValueOnce(10)

      .mockReturnValueOnce("x")

      .mockReturnValue("x");

    console.log(myMock(), myMock(), myMock(), myMock());

    // > 10, 'x', true, true

    test("filterTest", () => {

      const filterTestFn = jest.fn();

      // Faire en sorte que la simulation renvoie `true` pour le premier appel,

      // et `false` pour le second appel

      filterTestFn.mockReturnValueOnce(true).mockReturnValueOnce(false);

      const result = [11, 12].filter((num) => filterTestFn(num));

      // > [11]

      console.log(filterTestFn.mock.calls[0][0]); // 11

      console.log(filterTestFn.mock.calls[1][0]); // 12

    });

  });

1. **Modules de simulation Moking Modules**

Le mockage de modules fait référence à la pratique de substituer ou simuler des modules ou dépendances lors du processus de test. Dans les frameworks de test JavaScript tels que Jest, les modules peuvent être mockés pour remplacer leurs implémentations réelles par des versions simulées. Cela permet d'avoir un plus grand contrôle sur le comportement des modules et facilite les tests isolés de composants ou de fonctions spécifiques.

User.js

const axios = require("axios").default;

class Users {

  static all() {

    return axios

      .get("https://jsonplaceholder.typicode.com/posts/1")

      .then((resp) => resp.data);

  }

}

module.exports = Users;

**mock.test.js**

describe("Modules de simulation Moking Modules", () => {

    const axios = require("axios");

    const Users = require("../src/users");

    jest.mock("axios");

    test("doit récupérer les utilisateurs", () => {

      const users = [{ name: "Bob" }];

      const resp = { data: users };

      // axios.get.mockResolvedValue(resp);

      // ou vous pouvez utiliser ce qui suit en fonction de votre cas d'utilisation :

      axios.get.mockImplementation(() => Promise.resolve(resp));

      return Users.all().then((data) => expect(data).toEqual(users));

    });

  });

1. **Simulation partielle**

Le mock partiel fait référence à une technique de mockage où seule une partie spécifique d'un module est remplacée par une implémentation simulée, tandis que le reste du module conserve son comportement d'origine.

**foo-bar-baz.js**

const foo = "foo";

const bar = () => "bar";

const baz = () => "baz";

module.exports = { foo, bar };

module.exports.default = baz;

**mock.test.js**

 describe("Mock partielle", () => {

    const { bar, foo } = require("../src/foo-bar-baz");

    const defaultExport = require("../src/foo-bar-baz").default;

    jest.mock("../src/foo-bar-baz", () => {

      const originalModule = jest.requireActual("../src/foo-bar-baz");

      //Simule l'exportation par défaut et l'exportation nommée 'foo'

      return {

        \_\_esModule: true,

        ...originalModule,

        default: jest.fn(() => "mocked baz"),

        foo: "mocked foo",

      };

    });

    test("devrait faire une simulation partielle", () => {

      const defaultExportResult = defaultExport();

      expect(defaultExportResult).toBe("mocked baz");

      expect(defaultExport).toHaveBeenCalled();

      expect(foo).toBe("mocked foo");

      expect(bar()).toBe("bar");

    });

  });

1. **Implémentation simulées**

Pourtant, il existe des cas où il est utile d'aller plus loin que la possibilité de spécifier des valeurs de retour et de remplacer complètement l'implémentation d'une fonction simulée. Cela peut être fait avec jest.fn ou la méthode mockImplementationOnce sur les fonctions simulées.

Foo.js

module.exports = function () {

  // une implémentation;

};

describe("Mock Implementations (Implémentations simulées)", () => {

    jest.mock("../src/foo");

    const foo = require("../src/foo");

    // foo est une fonction simulée

    foo.mockImplementation(() => 42);

    foo();

    // > 42

    // Lorsque vous devez recréer un comportement complexe d'une fonction simulée, de sorte que plusieurs appels de fonction produisent des résultats différents, utilisez la méthode mockImplementationOnce :

    const myMockFn = jest

      .fn()

      .mockImplementationOnce((cb) => cb(null, true))

      .mockImplementationOnce((cb) => cb(null, false));

    myMockFn((err, val) => console.log(val));

    // > true

    myMockFn((err, val) => console.log(val));

    // > false

    // Lorsque la fonction simulée manque d'implémentations définies avec mockImplementationOnce, elle exécutera l'implémentation par défaut définie avec jest.fn (si elle est définie) :

    const myMockFn2 = jest

      .fn(() => "par défaut")

      .mockImplementationOnce(() => "premier appel")

      .mockImplementationOnce(() => "second appel");

    console.log(myMockFn2(), myMockFn2(), myMockFn2(), myMockFn2());

    // > 'premier appel', 'second appel', 'par défaut', 'par défaut'

    // Pour les cas où nous avons des méthodes qui sont typiquement enchaînées (et donc doivent toujours retourner this), nous avons une API sucrée pour simplifier cela sous la forme d'une fonction .mockReturnThis() qui se trouve également sur tous les simulations :

    const myObj = {

      myMethod: jest.fn().mockReturnThis(),

    };

    // est identique à

    const otherObj = {

      myMethod: jest.fn(function () {

        return this;

      }),

    };

  });