Jest And ReactTestingLibrary

npm install jest

Now that we have Jest installed, let’s create a file to write our tests. The Jest API looks for files that are either inside of a **\_\_tests\_\_/** directory or any file that ends in either **.test.js** or **.specs.js**.

jest \_\_tests\_\_/language\_spoken.test.js

**Configuring Jest**

Before we move on to actually writing our tests, we should cover a few best practices.

By default, each test produces the terminal output that we saw in the previous exercise. Jest allows us to customize this output by using command line flags. Though there are many [command-line flags](https://jestjs.io/docs/cli#reference), one of the most commonly used is the --coverage flag:

jest \_\_tests\_\_/ --coverage

This --coverage flag allows us to get a report of which lines of our code were actually tested. In addition to being outputted in the terminal, this report becomes available in a directory named **coverage/** that is created at runtime.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## Unit Testing with Jest (Part 2)

With our test() container set up, it is time to finish our first unit test by writing assertions to validate the various features of our code. To do this, Jest provides the expect() function.

The expect() function asserts how we expect our program to run and is used every time that we want to write a test. However, this function is rarely used alone – it can almost always be found in conjunction with matcher methods, like .toBe(), in the example below:

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## Matcher Functions

Congrats on writing your first unit test! In the last exercise, you used the expect() assertion function along with the .toEqual() matcher method. Let’s learn about a few more common matcher methods.

Take a look at the file below where we’ve now added a number of new assertions to test the getIngredients() method from the recipes module.

Let’s go over the matchers used in this example:

1. .toBeDefined() is used to verify that a variable is not undefined. This is often the first thing checked.
2. .toEqual() is used to perform deep equality checks between objects.
3. .toBe() is similar to .toEqual() but is used to compare primitive values.
4. .toBeTruthy() is used to verify whether a value is truthy or not.
5. .not is used before another matcher to verify that the opposite result is true
6. .toContain() is used when we want to verify that an item is in an array. In this case, since the .not matcher is used, we are verifying that "Ice Cream" is NOT in the array.

## Testing Async Code with Jest (Part 1)

Armed with how to write simple unit tests, we will now add a layer of complexity by exploring how to test asynchronous code in Jest.

Let’s return to the findRecipes() function from the recipes module. Remember, the findRecipes() method will make an asynchronous [REST API](https://www.codecademy.com/articles/what-is-rest) call and pass the resolved data to a callback function. We might use this function to find the ingredients for pesto like so:

If we wanted to make sure that this function does in fact get the requested data, we may be tempted to put our expect() assertion inside the callback function like so:

Une image contenant texte, capture d’écran, affichage, Police

Description générée automatiquement

Running this test will produce a bewildering pass! Again, Jest has no way to know that the callback is asynchronous so it will not wait for it and it will not see the failing expect() assertion.

To fix this issue, Jest allows us to add a done parameter in the test() callback function. The value of done is a function and, when included as a parameter, Jest knows that the test should not finish until this done() function is called.

Let’s take a final look at the test for the findRecipe() function, this time using the done parameter.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Let’s break down this example:

* In the first line of code, the done parameter is added to the callback passed to test(). Jest now knows to wait until that function is called before concluding the test.
* The done() function is called after the expect() assertion is made. This way, the expect() is guaranteed to be seen and any false-positives will be caught.

You should notice that the expect() and done() call are being made in a try block. Without this, if the assertion were to fail, expect() would throw an error before the done() function gets a chance to be called. From Jest’s perspective, the reason for the test failure would be a timeout error (since done() was never called) rather than the actual error thrown by the failed expect() assertion.

By using a catch block, we can capture the error value thrown and pass it to done(), which then displays it in the test output. Though not required, this is a best practice and will yield better test outputs.

++++++++++++++++++++++++++++++++++++++++++++++++=============+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++===============++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++====================+++++++++++++++++++++++++++++++++++++++++++++++++++++++============++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=====================++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++===================++++++++++++++++++++++++++++++++++++++++++++++++++++++==================+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++================++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=====++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++==================+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++================+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=========+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++==========++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++====++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++========================+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=============+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++===============++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++====================+++++++++++++++++++++++++++++++++++++++++++++++++++++++============++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=====================++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++===================++++++++++++++++++++++++++++++++++++++++++++++++++++++==================+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++================++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=====++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++==================+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++================+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=========+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++==========++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++====++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++========================+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Jest Documentation**

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. Jest est conçu pour être simple à configurer et à utiliser, tout en offrant des fonctionnalités puissantes pour écrire des tests robustes et fiables.

# **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**

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

# **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.

Haut du formulaire

# **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.

**Une image contenant texte, Appareils électroniques, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, affichage

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, affichage

Description générée automatiquement**

**Mock Functions (Fonctions simulées)**

Les fonctions simulées permettent de tester les liens entre le code en effaçant l'implémentation réelle d'une fonction, en capturant les appels à la fonction (et les paramètres passés dans ces appels), en capturant les instances des fonctions constructrices lorsqu'elles sont instanciées avec new, et en permettant la configuration des valeurs de retour au moment du test.

Il existe deux façons de simuler des fonctions : soit en créant une fonction simulée à utiliser dans le code de test, soit en écrivant une [simulation manuelle](https://jestjs.io/fr/docs/manual-mocks) pour remplacer une dépendance du module.

Example :

  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);

    });

  });

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, Site web

Description générée automatiquement**

**Propriété .mock**

Toutes les fonctions simulées ont cette propriété spéciale .mock, qui est l'endroit où sont conservées les données sur la façon dont la fonction a été appelée et ce que la fonction a renvoyé. La propriété .mock trace également la valeur de this pour chaque appel, il est donc possible de l'inspecter également :

 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");

    });

  });

## Valeurs de retour simulées

Les fonctions simulées peuvent également être utilisées pour injecter des valeurs de test dans votre code pendant un test :

Les fonctions simulées sont également très efficaces dans le code qui utilise un « style de passage de continuation » (NdT continuation-passing style) fonctionnel. Le code écrit dans ce style permet d'éviter le recours à des blocs compliqués qui recréent le comportement du composant réel qu'ils remplacent, en faveur de l'injection de valeurs directement dans le test juste avant leur utilisation.

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

    });

La plupart des exemples réels impliquent en fait de saisir une fonction fictive sur un composant dépendant et de la configurer, mais la technique est la même. Dans ces cas, essayez d'éviter la tentation d'implémenter la logique à l'intérieur de toute fonction qui n'est pas directement testée.

## Modules de simulation (Mocking Modules)

## Supposons que nous ayons une classe qui récupère les utilisateurs de notre API. La classe utilise [axios](https://github.com/axios/axios" \t "_blank) pour appeler l'API puis retourne l'attribut data qui contient tous les utilisateurs :

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;

Maintenant, afin de tester cette méthode sans toucher à l'API (et donc créer des tests lents et fragiles), nous pouvons utiliser la fonction jest.mock(...) pour simuler automatiquement le module axios.

Une fois le module simulé, nous pouvons fournir un mockResolvedValue pour .get qui renvoie les données que nous voulons que notre test vérifie. En effet, nous voulons que axios.get('/users.json') renvoie une fausse réponse.

describe("Modules de simulation Modl Mocks", () => {

    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));

    });

  });

## Simulation partielle (Mocking Partials[​](https://jestjs.io/docs/mock-functions" \l "mocking-partials" \o "Direct link to Mocking Partials))

Les sous-ensembles d'un module peuvent être simulés et le reste du module peut conserver son implémentation réelle :

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");

    });

  });

## Mock Implementations (Implémentations 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.

 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;

      }),

    };

  });

## Mock Names[​](https://jestjs.io/docs/mock-functions#mock-names)

## You can optionally provide a name for your mock functions, which will be displayed instead of 'jest.fn()' in the test error output. Use [.mockName()](https://jestjs.io/docs/mock-function-api/#mockfnmocknamename) if you want to be able to quickly identify the mock function reporting an error in your test output.

const myMockFn = jest

  .fn()

  .mockReturnValue('default')

  .mockImplementation(scalar => 42 + scalar)

  .mockName('add42');

## Custom Matchers Comparateurs personnalisés[​](https://jestjs.io/fr/docs/mock-functions#comparateurs-personnalis%C3%A9s) ()

## Enfin, pour qu'il soit moins difficile de déterminer comment les fonctions simulées ont été appelées, nous avons ajouté quelques fonctions comparatrices personnalisées :

// La fonction simulée a été appelée au moins une fois

expect(mockFunc).toHaveBeenCalled();

// La fonction simulée a été appelée au moins une fois avec les arguments spécifiés

expect(mockFunc).toHaveBeenCalledWith(arg1, arg2);

// Le dernier appel à la fonction simulée a été appelé avec les arguments spécifiés

expect(mockFunc).toHaveBeenLastCalledWith(arg1, arg2);

// Tous les appels et le nom de la simulation sont écrits en tant que snapshot

expect(mockFunc).toMatchSnapshot();

## Ces comparateurs sont du sucre pour les formes courantes d'inspection de la propriété .mock. Vous pouvez toujours le faire manuellement si cela vous convient mieux ou si vous avez besoin de quelque chose de plus spécifique :

// La fonction simulée a été appelée au moins une fois

expect(mockFunc.mock.calls.length).toBeGreaterThan(0);

// La fonction simulée a été appelée au moins une fois avec les arguments spécifiés

expect(mockFunc.mock.calls).toContainEqual([arg1, arg2]);

// Le dernier appel à la fonction fantaisie a été appelé avec les arguments spécifiés

expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1]).toEqual([

  arg1,

  arg2,

]);

// Le premier argument du dernier appel à la fonction simulée était `42`

// (notez qu'il n'existe pas d'aide sucrée pour cette assertion spécifique)

expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1][0]).toBe(42);

// Un snapshot vérifiera qu'une simulation a été appelée le même nombre de fois,

// dans le même ordre, avec les mêmes arguments. Il vérifiera également le nom.

expect(mockFunc.mock.calls).toEqual([[arg1, arg2]]);

expect(mockFunc.getMockName()).toBe('un nom de simulation');