Introduction à Jest pour les Tests

1. Introduction

Les tests d'applications sont un aspect important du développement Web. Cela permet d’éliminer les bogues avant que ces applications ne soient proposées aux utilisateurs. Tester les applications aide également à maintenir et à refactoriser le code, en particulier dans les projets collaboratifs dans lesquels un développeur apporte une modification hors contexte, favorisant ainsi une bonne collaboration entre ses coéquipiers. Il garantit également la qualité et l’exactitude du code.  
Le but est de montrer comment les tests unitaires sont effectués dans une application React-Vite avec Jest. De la configuration à la mise en œuvre.

<https://jestjs.io/>

1. Installation avec vite

* npm create vite@latest
* installation de Jest et Babel

npm install --save-dev jest jest-environment-jsdom jest-transform-stub @testing-library/jest-dom

npm install --save-dev babel-jest @babel/core @babel/preset-env @babel/preset-react

1. configuration de Babel

Il faut creer un fichier babel.config.cjs à la racine du projet.

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

Description générée automatiquement

1. Configurer Jest

Jest nécessite une certaine configuration pour fonctionner de manière transparente avec Vite et React. Créez un jest.config.cjs fichier à la racine de votre projet :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Cette configuration indique à Jest d'utiliser babel-jest pour transformer vos fichiers de test.

jest-transform-stub est un outil utile pour traiter des actifs non JavaScript dans les tests Jest. Il permet à Jest d'ignorer les importations d'actifs (comme les CSS, les images, etc.), qu'il ne peut pas gérer nativement, en les transformant en stub.

Jsdome st une implémentation purement JavaScript de nombreux standards Web, principalement les standards WHATWG DOM et HTML, à utiliser avec Node.js, permettant la simulation d'un environnement de navigateur pour tester le code JavaScript en dehors d'un navigateur.

## La nécessité de Babel dans les tests avec Vite et React

L'intégration de Babel dans notre configuration Jest joue un rôle crucial, notamment dans le contexte d'une application React construite avec Vite. Vite, de par sa conception, exploite les modules [ECMAScript](https://en.wikipedia.org/wiki/ECMAScript) (ESM) natifs pour une expérience de développement plus rapide, et prend automatiquement en charge les fonctionnalités JavaScript modernes et JSX. Cependant, Jest ne comprend pas nativement la syntaxe ESM ou JSX. C’est là que Babel devient incontournable.

Babel agit comme un pont entre le code JavaScript et JSX moderne que nous écrivons (et que Vite gère confortablement) et l'environnement JavaScript plus traditionnel dans lequel Jest opère. Lorsque nous exécutons nos tests, Jest invoque Babel pour transpiler le code. Cette étape de transpilation convertit JSX en appels de fonction JavaScript réguliers et transforme la syntaxe ES6+ en un format que Jest peut traiter.

Sous le capot, Babel utilise les préréglages spécifiés - @babel/preset-envet @babel/preset-react. Le @babel/preset-envpréréglage permet à Babel de transpiler la syntaxe ES6+ (comme les fonctions fléchées, les littéraux de modèles, etc.) jusqu'à ES5, garantissant ainsi la compatibilité avec l'environnement d'exécution de Jest. Le @babel/preset-react, quant à lui, traite spécifiquement de JSX, le transformant en React.createElementappels, qui est du JavaScript standard compris par Jest.

Sans Babel, Jest rencontrerait une syntaxe qu'il ne comprend pas (comme les instructions d'importation ou JSX), entraînant des erreurs de syntaxe et des échecs de tests. Par conséquent, Babel n’est pas seulement une commodité ; c'est une nécessité pour combler le fossé entre l'expérience de développement moderne fournie par Vite et les capacités de test de Jest. En intégrant Babel, nous garantissons que notre base de code React moderne et efficace reste testable et robuste, adhérant aux meilleures pratiques en matière de développement logiciel.

1. Ecrire son premier test

Ajouter dans scripts de package.json :

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

Description générée automatiquement

Pour executer un test : npm test

Pour tester la couverture de code : npm coverage

* Premier Test de App

Dans un dossier test, on rajoute un fichier : App.test.jsx

Ici on test :

Render pour recuperer le render de App et on test l’existence d’un element avec un text « Click »

expect permet de tester et la méthode toBeInTheDocument() teste la présence.

Si cela renvoi true, c’est que le test est ok

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Autre exemple d’un test simple :

Ceci test la sortie d’une fonction en fonction d’une entrée.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Describe permet de regrouper des tests et test et le test unitaire en lui-même.

Describe peut contenir plusieurs tests unitaires.

Avec except il existe plusieurs matchers disponibles :

* Vérité[​](https://jestjs.io/docs/using-matchers#truthiness)

Dans les tests, vous devez parfois faire la distinction entre undefined, null, et false, mais vous ne souhaitez parfois pas les traiter différemment. Jest contient des aides qui vous permettent d'être explicite sur ce que vous voulez.

* toBeNull correspond seulementnull
* toBeUndefined correspond seulementundefined
* toBeDefined est le contraire detoBeUndefined
* toBeTruthy correspond à tout ce qu'une if instruction considère comme vrai
* toBeFalsy correspond à tout ce qu'une if instruction considère comme faux
* Nombres

test('two plus two', () => {  
 const value = 2 + 2;  
 expect(value).toBeGreaterThan(3);  
 expect(value).toBeGreaterThanOrEqual(3.5);  
 expect(value).toBeLessThan(5);  
 expect(value).toBeLessThanOrEqual(4.5);  
  
 *// toBe and toEqual are equivalent for numbers*  
 expect(value).toBe(4);  
 expect(value).toEqual(4);  
});

* Chaines

test('there is no I in team', () => {  
 expect('team').not.toMatch(/I/);  
});  
  
test('but there is a "stop" in Christoph', () => {  
 expect('Christoph').toMatch(/stop/);  
});

* Tableaux et itérables

const shoppingList = [  
 'diapers',  
 'kleenex',  
 'trash bags',  
 'paper towels',  
 'milk',  
];  
  
test('the shopping list has milk on it', () => {  
 expect(shoppingList).toContain('milk');  
 expect(new Set(shoppingList)).toContain('milk');  
});

* Des exceptions

function compileAndroidCode() {  
 throw new Error('you are using the wrong JDK!');  
}  
  
test('compiling android goes as expected', () => {  
 expect(() => compileAndroidCode()).toThrow();  
 expect(() => compileAndroidCode()).toThrow(Error);  
  
 *// You can also use a string that must be contained in the error message or a regexp*  
 expect(() => compileAndroidCode()).toThrow('you are using the wrong JDK');  
 expect(() => compileAndroidCode()).toThrow(/JDK/);  
  
 *// Or you can match an exact error message using a regexp like below*  
 expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK$/); *// Test fails*  
 expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK!$/); *// Test pass*  
});