



formation.tech

Tests Automatisés sous React



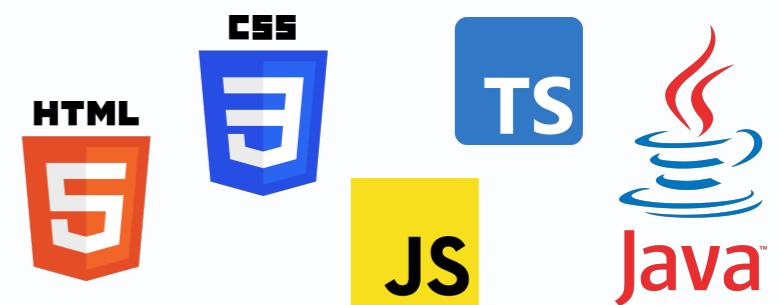
formation.tech

Introduction



Introduction - Formateur

- Romain Bohdanowicz
Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle
- Expérience
Formateur/Développeur Freelance depuis 2006
Près de 2000 jours de formation animées
- Langages
Expert : HTML / CSS / JavaScript / TypeScript / PHP / Java
Notions : C / C++ / Objective-C / C# / Python / Bash / Batch
- Certifications
PHP / Zend Framework / Node.js
- A propos
Premier site web à 12 ans (HTML/JS/PHP)
Triathlète du dimanche



Introduction - Horaires



- Matin
 - 9h - 10h
 - 10h15 - 11h15
 - 11h30 - 12h30
- Après-midi
 - 13h45 - 14h45
 - 15h - 16h
 - 16h15 - 17h15
- Questionnaire de satisfaction à remplir en fin de formation :
<https://stagiaire.formation.tech/>

Introduction - formation.tech



- Organisme de formation depuis 2016
- Référencé DataDock
- Certifié Qualiopi
- 15 formations au catalogue
- Une dizaine de formateurs indépendants
- Formations en français ou anglais
- <https://formation.tech/>



Introduction - WeAreDevs



- Studio de développement créé en 2017
- 1 salarié développeur senior
- Principales références
 - Cinexpert / Adeum
 - Sponsorise.me
 - Intel
 - Staytuned
 - STMicroelectronics
- <https://wearedevs.fr/>



Introduction - Et vous ?



- Pré-requis ?
- Rôle dans votre société ?
- Intérêt / objectif de cette formation ?



formation.tech

Tests Automatisés

Tests Automatisés - Introduction



- Comment tester son code ?
 - Manuellement : une personne effectue les tests
 - Automatiquement : les tests ont été programmés
- Historique
 - à partir de 1989 en Smalltalk et le framework SUnit
 - à partir de 1997 en Java avec JUnit
 - à partir de 2004 dans le navigateur avec Selenium

Tests Automatisés - Pourquoi ?



- Pourquoi automatiser les tests ?
 - plus l'application grandit, plus le risque d'introduire une régression est grand
ex: modifier une fonction qui est partagé par différentes
 - tester manuellement à chaque itération prendra à terme plus de temps qu'écrire le code du test
 - les tests automatisés peuvent se lancer sur différentes plate-formes et navigateurs très simplement
 - les tests aident à la compréhension du code, les lire permet de comprendre des comportements qui n'ont pas toujours été documentés
- Pourquoi continuer de tester manuellement ?
 - certains tests peuvent être simple à faire manuellement mais compliqués à automatiser (drag-n-drop...)
 - automatiser permet d'avoir accès à des choses inaccessibles manuellement (bouton caché par une popup...)



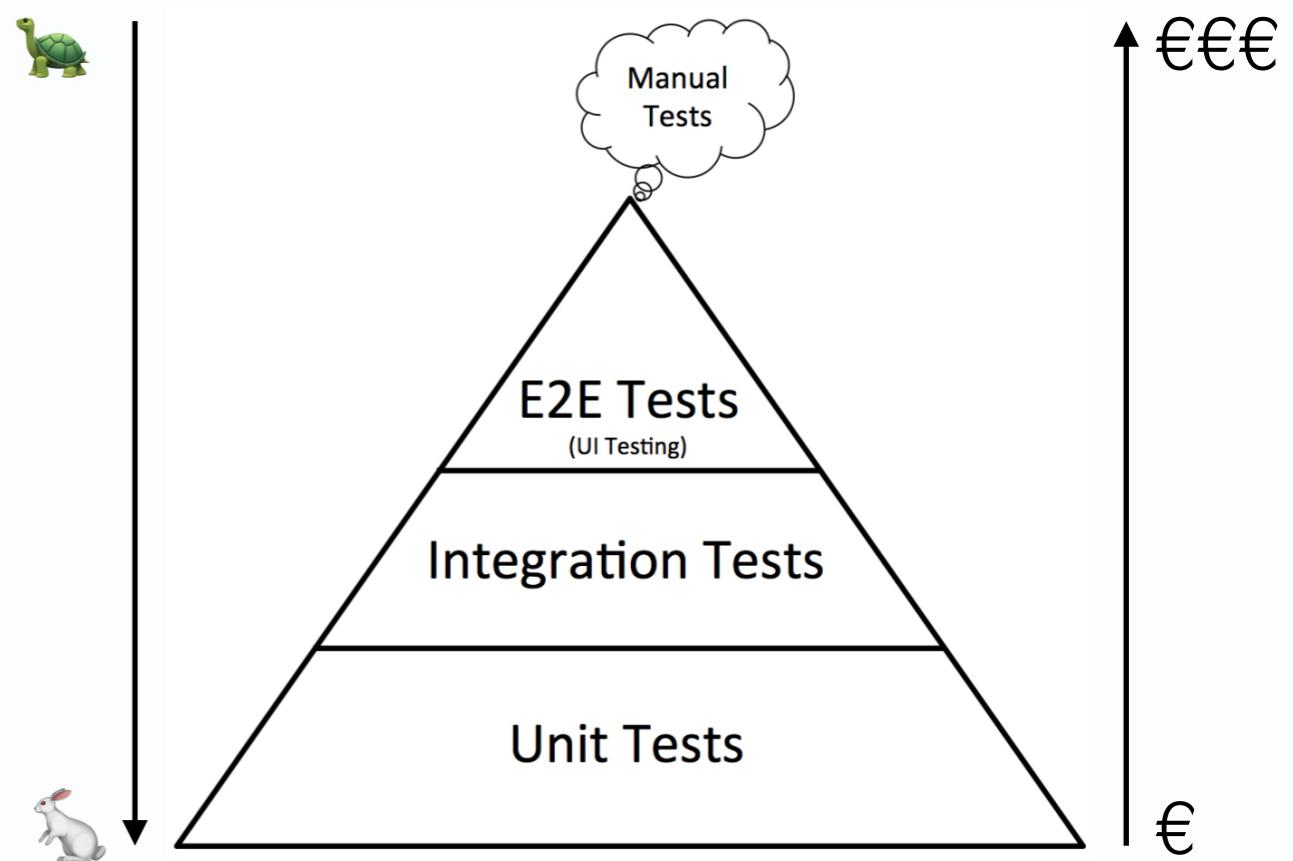
Tests Automatisés - Types de tests

- Types de tests :
 - tests de code statiques / linters
 - tests de code dynamiques / tests unitaires...
 - tests de déploiement
 - tests de sécurité (pentests)
 - tests de montée en charge
 - ...



Tests Automatisés - Pyramide des tests

- 3 types de tests automatisés au niveau code côté Front :
 - Test unitaire
Permet de tester les briques d'une application (classes / fonctions)
 - Test d'intégration
Teste que les briques fonctionnent correctement ensembles
 - Test End-to-End (E2E)
Vérifie l'application dans le client
- Une pyramide
 - plus le test est haut plus il est lent
 - plus le test est haut plus il coûte cher





Tests Automatisés - Quand exécuter ?

- Quand exécuter ?
 - tout le temps si on arrive à maintenir des tests performants (max 1-2 minutes)
 - avant un commit
 - avant un push
 - sur une plateforme d'intégration ou de déploiement continu (CI/CD)



Tests Automatisés - Organisation

- Ou placer ses tests ?
 - dans le même répertoire que le code testé
 - dans un répertoire *test* en préservant l'arborescence du répertoire *src*
 - dans un répertoire *test* sans lien avec l'arborescence



formation.tech

Jasmine

Jasmine - Expectations



- Dans Jasmine on utilise l'API expect pour vérifier le résultat du test ce qui lui donne un style phrasé :
`expect(true).toBe(true)`
- La fonction expect reçoit le code à tester, par exemple le retour d'une fonction
- La méthode expect retourne un objet matchers documenté ici :
<https://jasmine.github.io/api/edge/matchers.html>
- Les méthodes et propriétés principales de matchers :
 - `toBe` (équivalent à un `==`)
OK : `expect(1 + 2).toBe(3)`
KO : `expect(1 + 2).toBe('3')`
 - `not` (équivalent à un `!`)
OK : `expect(1 + 2).not.toBe(2)`
KO : `expect(1 + 2).not.toBe(3)`

Jasmine - Expectations



- toBeFalse / toBeTrue
OK : expect(false).toBeFalse()
KO : expect(0).toBeFalse()
- toBeFalsy (false après conversion) / toBeTruthy
OK : expect(0).toBeFalsy()
KO : expect(10).toBeFalsy()
- toBeDefined / toBeUndefined
- toBeNull / toBeNaN
- toEqual (deepEqual)
OK : expect({id: 3}).toEqual({id: 3}); // deepEqual
OK : expect({id: 3}).not.toBe({id: 3}); // ===
- toContain (string ou un tableau contient comparé avec deepEqual)
expect('ABC').toContain('C');
expect(['A', 'B', 'C']).toContain('C');
expect([{id: 3}]).toContain({id: 3});

Jasmine - Fonction pure



- Les fonctions pures sont les plus simples à tester :
 - elles sont prédictives, appelées avec les mêmes paramètres elles auront toujours le même retour
 - elles sont sans effets de bord (side-effect), elle n'appelle pas d'API externe (réseau, stockage...)
 - elles ne modifient pas leur paramètres d'entrée

```
export function sum(a, b) {  
  return Number(a) + Number(b);  
}
```

```
import { sum } from './sum';  
  
describe('sum function', () => {  
  it('should add positive number', () => {  
    expect(sum(1, 2)).toEqual(3);  
  });  
  it('should convert strings to numbers', () => {  
    expect(sum('1', '2')).toEqual(3);  
    expect(sum('2', '15')).toEqual(17);  
    expect(sum('5', '1')).toEqual(6);  
  });  
});
```

Jasmine - Tester les erreurs



- Parfois on peut simplement exécuter la fonction sans `expect`, si la fonction tombe sur le mot clé `throw` le test échoue
- Si on veut intercepter l'erreur (plus précis), on peut alors englober l'appel dans un callback et tester l'erreur avec `.toThrow` ou `.toThrowError`

```
function noEmptyArray(array) {
  if (!array.length) {
    throw new Error('array is empty');
  }
}

describe('noEmptyArray function', () => {
  it('should work well with filled array', () => {
    noEmptyArray(['A', 'B', 'C']);
  });
  it('should throw error with empty array', () => {
    expect(() => noEmptyArray([])).toThrow();
    expect(() => noEmptyArray([])).toThrow(new Error('array is empty'));
    expect(() => noEmptyArray([])).toThrowError('array is empty');
  });
});
```

Jasmine - Doubles



- Dans un test, un double est un morceau de code qui vient remplacer un autre morceau le temps du test
- Pourquoi remplacer un morceau de code par un autre ?
 - parce qu'on veut vérifier qu'il soit appelé correctement
 - parce qu'il est long à exécuter et qu'il ralenti les tests
 - parce qu'il dépend d'un API extérieur ou de variables globales qu'on a du mal à contrôler (requêtes AJAX, localStorage, base de données)
 - parce son résultat est aléatoire

Jasmine - Doubles



- › Théoriquement il existe plusieurs types de double
 - Fake : on réécrit une seconde version de notre fonction ou classe que l'on injectera dans l'application, mais on ne pourra pas vérifier simplement que les appels ont été fait
 - Dummy: on génère via une bibliothèque une fonction qui ne fait rien
 - Stub : on génère via une bibliothèque une fonction qui reproduit un comportement similaire à la fonction d'origine
 - Mock : on génère une fonction avec un comportement et on paramètre les expectations à la création (avant l'appel)
 - Spy : on génère une fonction avec un comportement et on paramètre les expectations à la fin du test (après l'appel)

Jasmine - Spy



- › Jasmine nous fourni un API pour créer des *spies*, via les méthodes
 - `jasmine.createSpy`
 - `spyOn`
 - `spyOnProperty`



Jasmine - jasmine.createSpy

- jasmine.createSpy va créer une fonction qui va enregistrer ses appels avec les paramètres
- On pourra vérifier en fin de test que la fonction a bien été appelée, le nombre d'appels, les paramètres

```
function doSomething(cb) {  
  cb('ABC');  
}  
  
describe('doSomething function', () => {  
  it('should call cb once', () => {  
    const spy = jasmine.createSpy();  
    doSomething(spy);  
    expect(spy).toHaveBeenCalled();  
    expect(spy).toHaveBeenCalledWith('ABC')  
    expect(spy).toHaveBeenCalledTimes(1);  
    expect(spy).toHaveBeenCalledOnceWith('ABC');  
  });  
});
```

Jasmine - spyOn



- spyOn va nous permettre d'écouter et de transformer les appels au niveau d'une méthode
- Voici un exemple où une fonction *thatCallMyObjMethod* va appeler une méthode d'une autre objet *MyObj.myMethod* :

```
const MyObj = {  
  myMethod(val) {  
    return val.toUpperCase();  
  }  
}  
  
function thatCallMyObjMethod() {  
  return MyObj.myMethod('abc');  
}  
  
describe('thatCallMyObjMethod function', () => {  
  it('should call MyObj.myMethod once', () => {  
    expect(thatCallMyObjMethod()).toBe('ABC');  
  });  
});
```

- Le test n'est pas unitaire, le test peut échouer à cause de la fonction imbriquée
- Si la fonction imbriquée est lente où dépend d'un API extérieur on pourrait avoir besoin de la modifier le temps du test

Jasmine - spyOn



- spyOn peut garder le code d'origine en enregistrant les appels avec *.and.callThrough()*

```
const MyObj = {
  myMethod(val) {
    return val.toUpperCase();
  }
}

function thatCallMyObjMethod() {
  return MyObj.myMethod('abc');
}

describe('doSomething function', () => {
  it('should call cb once', () => {
    spyOn(MyObj, 'myMethod').and.callThrough();
    expect(thatCallMyObjMethod()).toBe('ABC');
    expect(MyObj.myMethod).toHaveBeenCalledWith('abc');
  });
});
```



Jasmine - spyOn

- spyOn se transformer en fonction vide avec .and.stub()

```
const MyObj = {  
  myMethod(val) {  
    return val.toUpperCase();  
  }  
}  
  
function thatCallMyObjMethod() {  
  return MyObj.myMethod('abc');  
}  
  
describe('doSomething function', () => {  
  it('should call cb once', () => {  
    spyOn(MyObj, 'myMethod').and.stub();  
    expect(thatCallMyObjMethod()).toBeUndefined();  
    expect(MyObj.myMethod).toHaveBeenCalledWith('abc');  
  });  
});
```

Jasmine - spyOn



- Le code généré par `spyOn` se détruit automatiquement à la fin du test (du `it` ou du `describe` selon où `spyOn` a été appelé)

```
const MyObj = {
  myMethod(val) {
    return val.toUpperCase();
  }
}

function thatCallMyObjMethod() {
  return MyObj.myMethod('abc');
}

describe('doSomething function', () => {
  it('should call cb once', () => {
    spyOn(MyObj, 'myMethod').and.stub();
    expect(thatCallMyObjMethod()).toBeUndefined();
    expect(MyObj.myMethod).toHaveBeenCalledOnceWith('abc');
  });
  it('should call cb once', () => {
    expect(thatCallMyObjMethod()).toBe('ABC');
  });
});
```

Jasmine - spyOn



- On peut écrire une toute nouvelle fonction le temps du test avec :
 - .and.returnValue pour choisir la valeur de retour
 - .and.returnValues pour choisir les valeurs de retour dans le cas d'appels multiples
 - .and.throwError pour lancer une erreur
 - .and.resolveTo ou .and.rejectWith pour retourner des promesses
 - .and.callFake pour écrire sa propre implémentation

```
describe('doSomething function', () => {
  it('should call cb once', () => {
    spyOn(MyObj, 'myMethod').and.returnValue('DEF');
    expect(thatCallMyObjMethod()).toBe('DEF');
    expect(MyObj.myMethod).toHaveBeenCalledWith('abc');
  });
});

describe('doSomething function', () => {
  it('should call cb once', () => {
    spyOn(MyObj, 'myMethod').and.callFake(() => 'DEF');
    expect(thatCallMyObjMethod()).toBe('DEF');
    expect(MyObj.myMethod).toHaveBeenCalledWith('abc');
  });
});
```

Jasmine - Tester du code asynchrone



- Avec du code asynchrone le test se termine parfois avant que le code ait été appelé

```
export function asyncCallback(cb) {
  setTimeout(() => {
    cb()
  }, 1000);
}

import { asyncCallback } from './async-callback';

describe('asyncCallback function', () => {
  it('should call callback', () => {
    function cb() {
      // le test sera terminé avant l'appel du callback
      expect(true).toBe(false);
    }
    asyncCallback(cb)
  });
});
```

- Pour résoudre le problème on peut :
 - utiliser la fonction done de Jasmine
 - utiliser les promesses

Jasmine - Tester du code asynchrone



- Avec la fonction done

```
describe('asyncCallback function', () => {
  it('should call callback', (done) => {
    function cb() {
      // ce test échoue comme prévu
      expect(true).toBe(false);
      done();
    }
    asyncCallback(cb)
  });
});
```

Jasmine - Tester du code asynchrone



- Avec les promesses

```
function thatReturnPromise() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('ABC');
    }, 1000);
  });
}

describe('thatReturnPromise function', () => {
  it('should call Promise', () => {
    return thatReturnPromise().then((val) => {
      expect(val).toBe('ABC');
    });
  });
});
```

- Dans ce cas on pensera bien à retourner la promesse dans la fonction *it*



Jasmine - Tester du code asynchrone

- Comme le suggère VSCode, ce code peut se transformer avec la syntaxe async/await

```
describe('thatReturnPromise function', () => {
  it('should call Promise', () => {
    Convert to async function
    thatReturnPromise().then((val) => {
      expect(val).toBe('fff');
    });
  });
});
```

- Ce qui donne

```
describe('thatReturnPromise function', () => {
  it('should call Promise', async () => {
    const val = await thatReturnPromise();
    expect(val).toBe('ABC');
  });
});
```

Jasmine - Tester du code asynchrone



- › Contrôler le temps

```
function thatReturnPromise() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('ABC');
    }, 10_000);
  });
}

describe('thatReturnPromise function', () => {
  it('should call Promise in 10s', async () => {
    const val = await thatReturnPromise();
    expect(val).toBe('ABC');
  });
});
```

- › Ce type de code est problématique, on est obligé d'attendre 10 secondes avant le résultat (d'ailleurs le test va échouer car jasmine a un timeout de 5 secondes par test)



Jasmine - Tester du code asynchrone

- Avec `jasmine.clock` on peut contrôler le temps

```
describe('thatReturnPromise function', () => {
  beforeEach(() => {
    jasmine.clock().install();
  });
  afterEach(() => {
    jasmine.clock().uninstall();
  });
  it('should call Promise in 10s', async () => {
    const promise = thatReturnPromise();
    jasmine.clock().tick(10000);
    const val = await promise;
    expect(val).toBe('ABC');
  });
});
```

- La méthode `jasmine.clock().tick` permet d'avancer de n millisecondes
- Voir aussi `jasmine.clock().mockDate()` qui permet de définir la date de son choix.

Jasmine - Bonnes pratiques



- Vérifier que le test échoue :
Parfois les expect ne sont pas exécuter, vérifier donc avec un test comme
`expect(true).toBeFalsy()`
que la ligne a bien été exécutée et que le test échoue
- Utiliser TypeScript pour vérifier les problèmes de types
- Faire attention à ne pas créer de dépendance entre les tests, je dois pouvoir supprimer un test sans en mettre à jour d'autres (ou les exécuter dans un ordre aléatoire)
- Exécuter les tests plusieurs fois de suite de temps en temps pour détecter les tests qui passent aléatoirement (code asynchrone)
- Privilégier les fonctions :
 - qui sont appelées fréquemment (meilleure couverture de code)
 - dont le résultat est essentiel à l'application (total TTC d'une facture...)
 - qui ont une complexité avancée (de nombreux if, for...)
- Utiliser les doubles (`jasmine.createSpy, spyOn...`) pour isoler les tests de l'extérieur (API réseaux, de stockage...)



formation.tech

Jest

Jest - Introduction



- Framework de test créé en 2014 par Facebook
- Sous Licence MIT depuis septembre 2017
- Permet de lancer des tests :
 - unitaires / d'intégration (dans Node.js)
 - fonctionnels / E2E (via Puppeteer ou PlayWright)
- Peut s'utiliser avec ou sans configuration
- Les tests se lancent en parallèle dans les Workers Node.js
- Intègre par défaut :
 - Calcul de coverage (via Istanbul)
 - Mocks (natifs ou en installant Sinon.JS)
 - Snapshots



Jest - Installation

- › Installation

```
npm install --save-dev jest
```

```
yarn add --dev jest
```



Jest - Hello, world !

- Sans configuration, les tests doivent se trouver dans un répertoire `__tests__`, ou bien se nommer `*.test.js` ou `*.spec.js`

```
// src/hello.js
const hello = (name = 'World') => `Hello ${name} !`;

module.exports = hello;
```

```
// __tests__/hello.js
const hello = require('../src/hello');

test('Hello, world !', () => {
  expect(hello()).toBe('Hello World !');
  expect(hello('Romain')).toBe('Hello Romain !');
});
```



Jest - Lancements des tests

- › Si Jest localement
node_modules/.bin/jest
- › Si Jest globalement
jest
- › Avec un script test dans package.json
npm run test
npm test
npm t

```
// package.json
{
  "devDependencies": {
    "jest": "^22.0.6"
  },
  "scripts": {
    "test": "jest"
  }
}
```

```
MacBook-Pro:hello-jest romain$ node_modules/.bin/jest
PASS  __tests__/_hello.js
  ✓ Hello, world ! (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.701s, estimated 1s
Ran all test suites.
```



Jest - Watchers

- › En mode Watch

```
node_modules/.bin/jest --watchAll  
jest --watchAll  
npm t -- --watchAll
```

```
MacBook-Pro:hello-jest romain$ npm t -- --watchAll  
PASS  __tests__/hello.js  
PASS  __tests__/calc.js
```

```
Test Suites: 2 passed, 2 total  
Tests:       3 passed, 3 total  
Snapshots:   0 total  
Time:        0.65s, estimated 1s  
Ran all test suites.
```

Watch Usage

- › Press **f** to run only failed tests.
- › Press **o** to only run tests related to changed files.
- › Press **p** to filter by a filename regex pattern.
- › Press **t** to filter by a test name regex pattern.
- › Press **q** to quit watch mode.
- › Press **Enter** to trigger a test run.



Jest - Coverage

- Avec calcul du coverage
node_modules/.bin/jest --coverage
jest --coverage
npm t -- --coverage
- Par défaut le coverage s'affiche dans la console et génère des fichiers Clover, JSON et HTML dans le dossier coverage

```
MacBook-Pro:hello-jest romain$ npm t -- --coverage
PASS  __tests__/calc.js
PASS  __tests__/hello.js
```

```
Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.722s, estimated 1s
Ran all test suites.
```

File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Lines
All files	86.67	100	60	100	
calc.js	83.33	100	50	100	
hello.js	100	100	100	100	



Jest - Partager des variables entre les tests

- › On peut utiliser la portée de closure

```
describe('A suite is just a function', () => {
  let a;

  test('and so is a spec', () => {
    a = true;

    expect(a).toBe(true);
  });
});
```

Jest - Hooks



- Dans une suite de tests, certaines méthodes seront appelées automatiquement durant la vie du test
 - `beforeEach`, avant chaque test de la suite (avant chaque *it*)
 - `afterEach`, après chaque test
 - `beforeAll`, avant le premier test de la suite (avant le premier *it*)
 - `afterAll`, après le dernier test de la suite

```
describe('A suite with some shared setup', () => {
  let foo = 0;
  beforeEach(() => {
    foo += 1;
  });
  afterEach(() => {
    foo = 0;
  });
  beforeAll(() => {
    foo = 1;
  });
  afterAll(() => {
    foo = 0;
  });
});
```



Jest - Désactiver les tests

- › Pour désactiver certains tests on peut :
 - les commenter
 - utiliser la fonction `test.skip` au lieu de `test`
 - utiliser la fonction `describe.skip` au lieu de `describe`

```
describe('sum function', () => {
  test.skip('should add positive number', () => {
    expect(sum(1, 2)).toEqual(3);
  });
  test('should convert strings to numbers', () => {
    expect(sum('1', '2')).toEqual(3);
  });
});      ➔ hello-jasmine npm t

> @ test /Users/romain/Desktop/prepa-angular-tests/hello-jasmine
> jasmine

Randomized with seed 01882
Started
.

Ran 1 of 5 specs
1 spec, 0 failures
Finished in 0.011 seconds
```



Jest - Désactiver les tests

- › On peut également forcer l'exécution d'un test (et pas les autres) avec
 - *test.only* pour un test en particulier
 - *describe.only* pour un groupe de test



Jest - Tester les erreurs

- › Pour tester les erreurs on utilise
 - un callback dans le expect
 - toThrow

```
export function throwError() {
  throw new Error('Error Message');
}

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

test('throwError function', () => {
  expect(() => throwError()).toThrow();
  expect(() => throwError()).toThrow('Error Message');
  expect(() => throwError()).toThrow(/error message/i);
  expect(() => throwError()).toThrow(Error);
  expect(() => throwError()).toThrow(new Error('Error Message'));
});
```

Jest - Mocks



- Créer une fonction de test

```
export function withCallback(cb: (val: string) => void) {  
  cb('ABC');  
}
```

```
import { expect, test, vitest } from 'vitest';  
import { withCallback } from './withCallback';  
  
test('withCallback function', () => {  
  const mockFn = vitest.fn();  
  
  withCallback(mockFn);  
  
  expect(mockFn).toHaveBeenCalled();  
  expect(mockFn).toHaveBeenCalledOnce();  
  expect(mockFn).toHaveBeenCalledTimes(1);  
  expect(mockFn).toHaveBeenCalledWith('ABC');  
});
```



Jest - Mocks

- Avec une implémentation

```
export function withCallbackReturn(cb: (val: string) => string): string {  
  return cb('ABC');  
}
```

```
import { expect, test, vitest } from 'vitest';  
import { withCallbackReturn } from './withCallbackReturn';  
  
test('withCallbackReturn function', () => {  
  const mockFn = vitest.fn().mockImplementation(() => 'XYZ');  
  // en raccourci :  
  // const mockFn = vitest.fn().mockReturnValue('XYZ');  
  
  const val = withCallbackReturn(mockFn);  
  
  expect(mockFn).toHaveBeenCalled();  
  expect(mockFn).toHaveBeenCalledOnce();  
  expect(mockFn).toHaveBeenCalledTimes(1);  
  expect(mockFn).toHaveBeenCalledWith('ABC');  
  expect(val).toBe('XYZ');  
});
```

Jest - Mocks



- › Pour espionner une méthode d'un objet

```
export function secondsFromNow(timestamp: number) {
  return Date.now() - timestamp;
}
```

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

test('secondsFromNow function', () => {
  vitest.spyOn(Date, 'now').mockReturnValue(new Date(2023, 9, 1, 0, 0, 0, 30).getTime())

  expect(secondsFromNow(new Date(2023, 9, 1, 0, 0, 0).getTime())).toBe(30)
});
```

Jest - Mocks



- › Pour espionner un module

```
export async function fetchUsers() {  
  const res = await fetch('https://jsonplaceholder.typicode.com/users');  
  return await res.json();  
}
```

```
import { fetchUsers } from './fetchUsers';  
  
export async function listUsers() {  
  const users = await fetchUsers();  
  return users;  
}
```

```
import { expect, test, vitest } from 'vitest';  
import { listUsers } from './listUsers';  
import { fetchUsers } from './fetchUsers';  
  
vitest.mock('./fetchUsers');  
  
test('listUsers function', async () => {  
  vitest.mocked(fetchUsers).mockResolvedValue([{id: 1, name: 'Toto'}])  
  const users = await listUsers();  
  expect(users).toEqual([{id: 1, name: 'Toto'}])  
});
```



Jest - Tester les timers

- La fonction `jest.useFakeTimers()` transforme les timers (`setTimeout`, `setInterval...`) en mock

```
export function timeout(delay: number, arg: any) {
  return new Promise((resolve) => {
    setTimeout(resolve, delay, arg);
  });
}
```

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

// ✓ hello function 1002ms
test('hello function', async () => {
  const val = await timeout(1000, 'ABC');
  expect(val).toBe('ABC');
});

// ✓ hello function
test('hello function', async () => {
  vitest.useFakeTimers();
  const promise = timeout(1000, 'ABC');
  vitest.advanceTimersByTime(1000);
  const val = await promise;
  expect(val).toBe('ABC');
  vitest.useRealTimers();
});
```



formation.tech

Tests React

Tests React - Introduction



- Le plus simple pour exécuter ses tests avec React : vitest
- Vitest utilise le même API que Jest et supporte JSX et TypeScript par défaut



Tests React - Hello

- › Pour tester ses composants Jest/Vitest vont lancer les tests dans Node.js (via des Workers)
- › Il faut donc émuler les APIs Web via des bibliothèques comme JSDOM ou plus moderne Happy DOM
- › vitest --dom

```
// npx vitest --dom
test('Hello renders (ReactDOM)', () => {
  const rootEl = document.createElement('div');
  createRoot(rootEl).render(<Hello />);
});
```



Tests React - 3 façons de tester

- Plusieurs options pour tester unitairement :
 - Utiliser l'object document fourni par JSDOM / Happy DOM
 - Utiliser react-test-renderer qui va faire le rendu des composants sous forme d'objet literal
 - Utiliser des bibliothèques haut niveau comme Enzyme ou Testing Library (recommandé)

```
// npx vitest --dom
test('Hello renders (ReactDOM)', () => {
  const rootEl = document.createElement('div');
  createRoot(rootEl).render(<Hello />);
});

test('Hello renders (react-test-renderer)', () => {
  const renderer = ReactTestRenderer.create(
    <Hello />
  );
});

// npx vitest --dom
test('Hello renders (testing-library)', () => {
  render(<Hello />);
});
```



Tests React - Base

- › Pour tester un composant React il faut en faire le rendu

```
// src/App.test.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

- › 2 inconvénients ici :
 - Nécessite que document existe (exécuter les tests dans un navigateur ou utiliser des implémentations de document côté Node.js comme JSDOM)
 - Les composants enfants du composant testé seront également rendu, le test n'est pas un test unitaire mais un test d'intégration



Tests React - Snapshot Testing

- Facebook fourni un paquet npm pour simplifier les tests : react-test-renderer
- Ici on fait un simple Snapshot, c'est à dire une capture du rendu du composant, si lors d'un test futur le rendu est modifié le test échoue

```
import React from 'react';
import renderer from 'react-test-renderer';
import { Hello } from './Hello';

test('it renders like last time', () => {
  const tree = renderer
    .create(<Hello />)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```



Tests React - Shallow Rendering

- On peut également faire appel à ShallowRenderer qui ne va faire qu'un seul niveau de rendu, et donc rendre le test unitaire

```
// src/App.test.js
import React from 'react';
import App from './App';
import ShallowRenderer from 'react-test-renderer/shallow';
import { Hello } from './Hello';
import { CounterButton } from './CounterButton';

it('renders without crashing', () => {
  const renderer = new ShallowRenderer();
  renderer.render(<App />);
  const result = renderer.getRenderOutput();

  expect(result.type).toBe('div');
  expect(result.props.children).toEqual([
    <Hello firstName="Romain" />,
    <hr />,
    <CounterButton/>,
  ]);
});
```



Tests React - Enzyme

- Facebook recommande également l'utilisation de la bibliothèque Enzyme, créée par AirBnB.
- Elle fourni un API haut niveau (proche de jQuery) pour manipuler les tests des composant

```
import React from 'react';
import { Hello } from './Hello';
import { shallow } from 'enzyme';

test('it renders without crashing with enzyme', () => {
  shallow(<Hello />);
});

test('it renders without crashing with enzyme', () => {
  const wrapper = shallow(<Hello />);
  expect(wrapper.contains(<div>Hello !</div>)).toEqual(true);
});

test('it renders without crashing with enzyme', () => {
  const wrapper = shallow(<Hello firstName="Romain"/>);
  expect(wrapper.contains(<div>Hello Romain !</div>)).toEqual(true);
});
```

Tests React - Tester des événements



```
import React, { Component } from 'react';

export class CounterButton extends Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({
      count: this.state.count + 1,
    });
  }

  render() {
    return (
      <button onClick={this.handleClick}>{this.state.count}</button>
    );
  }
}
```



Tests React - Tester des événements

- Avec Enzyme :

```
import React from 'react';
import { CounterButton } from './CounterButton';
import { shallow } from 'enzyme';

test('it renders without crashing', () => {
  shallow(<CounterButton />);
});

test('it contains 0 at first rendering', () => {
  const wrapper = shallow(<CounterButton />);
  expect(wrapper.text()).toBe('0');
});

test('it contains 1 after click', () => {
  const wrapper = shallow(<CounterButton />);
  wrapper.simulate('click');
  expect(wrapper.text()).toBe('1');
});
```



Tests React - Tester des événements

- Avec Testing Library :

```
import { render, screen, fireEvent } from "@testing-library/react";
import { CounterButton } from "./Counter";
import { expect, test } from "vitest";

// npx vitest --dom
test('CounterButton (testing-library)', () => {
  render(<CounterButton />);
  const button = screen.getByRole('button');

  expect(button.innerText).toContain('0');

  fireEvent.click(button);

  expect(button.innerText).toContain('1');
});
```