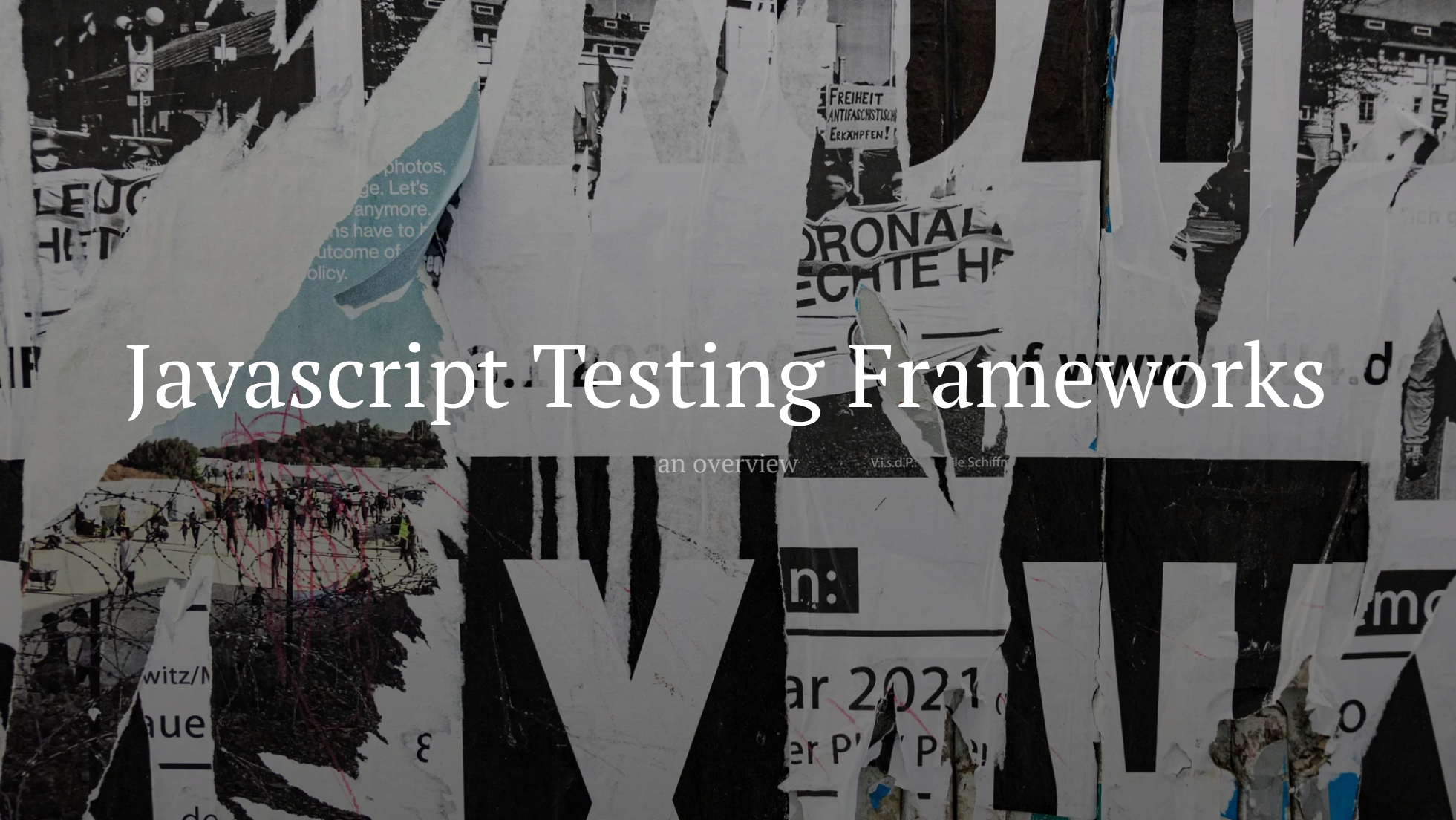


Javascript Testing Frameworks

an overview

Tester pour prévenir…

- une erreur du développeur introduit …
  - Une erreur est une décision inappropriée ou erronée, faite par un développeur, qui conduit à l'introduction d'un défaut.
- un défaut dans le système qui provoquera …
  - Un défaut est une imperfection dans un des aspects du système qui contribue, ou peut potentiellement contribuer, à la survenance d'une ou de plusieurs défaillances
  - Parfois, il faut plusieurs défauts pour causer une défaillance.
- sa défaillance à l'exécution.
  - Une défaillance est un comportement inacceptable présenté par un système.
  - La fréquence des défaillances reflète la fiabilité.


On a bien reçu votre livraison, par contre on n'a pas trouvé les tests
Les tests ?
Quels tests ?
Bah, les tests unitaires... Pour contrôler l'applicatif, la QA, tout ça...
Ah oui, les tests unitaires...
Y a pas de tests unitaires
Ah bon, y a pas de tests ? C'est pourtant la norme...
Comment vous assurez-vous que tout fonctionne alors ?
Ben...
IT'S MAGIC
CommitStrip.com

Le test : une première définition ...

> Le test est un processus manuel ou automatique, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification

Extrait de la norme IEEE-STD729, 1983.

Le test

Essayer pour trouver des bugs

Le test

Essayer pour voir si ça marche.

Le test

trouver des bugs

Essayer pour voir si ça marche.

- Apprendre
  - pourquoi c'est fait
  - ce que ça doit faire
  - comment c'est fait
  - comment ça marche
- Modéliser
- S'en faire une idée
- Exécuter
- Analyser

- Qu'y a-t-il à **voir**?
- Que faut-il regarder?
- Qu'est-ce qui est visible?
- Qu'est ce qu'on cherche?
- Comment le regarder?

- Qu'est ce qui devrait marcher?
- Identifier une erreur
- Diagnostiquer une erreur
- Catégoriser ces erreurs

Qu’est-ce qu’on teste? quelles propriétés?

- Fonctionnalité
- Sécurité / intégrité
- Utilisabilité
- Cohérence
- Maintenabilité
- Efficacité
- Robustesse
- Sûreté de fonctionnement
- Etc.

Comment on teste?

- Test statique
  - relecture / revue de code
  - analyse automatique (vérification de propriétés, règles de codage…)
- Test dynamique
  - on exécute le programme avec des valeurs en entrée et on observe le comportement

Comment on teste?

- Test fonctionnel (test boîte noire)
  - Utilise la description des fonctionnalités du programme
- Test structurel (test boîte blanche)
  - Utilise la structure interne du programme

Avec quoi on teste?

Une spécification: exprime ce qu'on attend du système

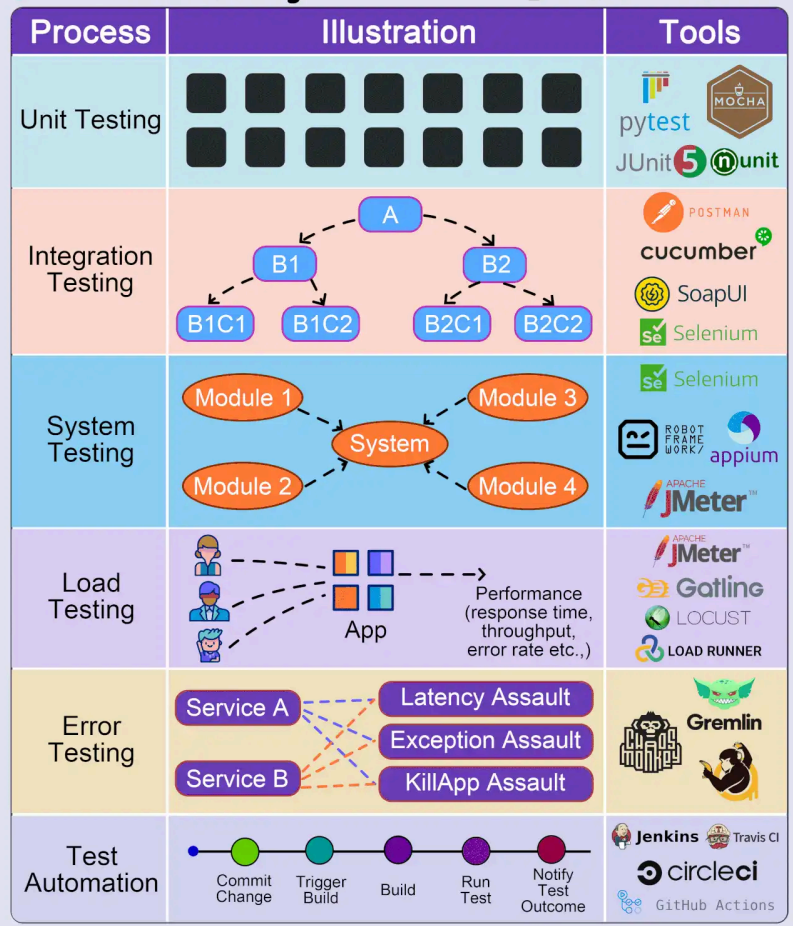
- des règles de codage
- un cahier des charges (en langue naturelle)
- commentaires dans le code
- contrats sur les opérations (à la Eiffel)
- un modèle UML
- une spécification formelle (automate, modèle B…)

Différents types de tests

1. Test unitaire, *Unit Testing: Ensures individual code components work correctly in isolation*
2. Test d'intégration, *Integration Testing: Verifies that different system parts function seamlessly together*
3. Test système, *E2E testing, System Testing: Assesses the entire system's compliance with user requirements and performance*
4. Test de charge, *Load Testing: Tests a system's ability to handle high workloads and identifies performance issues.*
5. *Error Testing: Evaluates how the software handles invalid inputs and error conditions*
6. Non regression, *Test Automation: Automates test case execution for efficiency, repeatability, and error reduction*
7. Test de sécurité (Pen testing, …)

Best Ways To Test System Functionality

blog.bytebytego.com



Et dans le monde du front

MochaJS

> Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun.

- Provides compatibility for both frontend and backend testing
- NodeJS debugger is supported which makes error tracing easier
- Accurate reporting
- Provides support for all browsers including the headless Chrome library
- Very convenient framework for the developers to write test cases

https://mochajs.org/

JEST

> JEST is arguably the most popular JavaScript testing framework used and maintained by Facebook.

- Compatible with NodeJS, React, Angular, VueJS, and other Babel based projects
- Standard syntax with documentation support
- Very fast and highly performant
- Managing tests with larger objects is possible using Live Snapshots

https://jestjs.io/

Jasmine

Introduced in 2010, Jasmine is an open-source JavaScript testing framework. It is capable of testing all types of JavaScript applications. This framework supports Behavioral Driven Development (BDD).

Using Jasmine, one can perform test cases similar to user behavior on a website.

- Provides small, clean and straightforward syntax for easy testing
- Does not require any Document Object Model (DOM)
- Provides support for both frontend and backend tests
- Ease in coding as the syntax used is very similar to a natural language
- Strong documentation and community support

https://jasmine.github.io/

Karma

Karma is another popular open-source productive testing environment. It allows a QA to perform tests for an application in different environments. Karma allows the application script to be executed on real browsers and devices like phones and tablets.

- Supports integrations with top CI/CD tools like Jenkins, Travis, and Semaphore
- Tests on real devices and browsers are possible. Want to run a quick automated test on a real device cloud? Try now.
- Provides support for headless environments like PhantomJS
- Supports remote testing directly from a terminal or IDE
- Is framework agnostic, which means one can describe tests with popular frameworks like Mocha, Jasmine. One can also write a simple adapter for a specific framework.

https://karma-runner.github.io/3.0/

Cypress

Fast, easy and reliable testing for anything that runs in a browser.

A complete end-to-end testing framework.

- open source and popular
- Supports integrations with top CI/CD tools like Jenkins, Travis, and Semaphore
- great dashboard

https://www.cypress.io/

Jest into an Angular application and library

a demo

Why JEST

- Sensible faster; parallelized test runs
- Snapshot testing; to make sure your UI does not change unexpectedly
- Rich CLI options; only run failed tests, filter on filename and/or test name, only run related tests since the latest commit
- Readable and useful tests reports
- Sandboxed tests; which means automatic global state resets
- Built in code coverage

Creating an angular application

Setup a demo app

```
ng new my-awesome-app
```

Start the app

```
cd my-awesome-app
npm start
```

Run the tests

```
npm test
```

```
npm install -D jest jest-preset-angular @types/jest
```

Configuration 1/3

In your project root, create `setup-jest.ts` file with following contents:

```
import 'jest-preset-angular/setup-jest';

Object.defineProperty(window, 'CSS', { value: null });
Object.defineProperty(window, 'getComputedStyle', {
  value: () => {
    return {
      display: 'none',
      appearance: ['-webkit-appearance'],
    };
  },
});
Object.defineProperty(document, 'doctype', {
  value: '<!DOCTYPE html>',
});
Object.defineProperty(document.body.style, 'transform', {
  value: () => {
    return {
      enumerable: true,
      configurable: true,
    };
  },
});
```

Configuration 2/3

Add the following section to your root `package.json`

```
{
  "jest": {
    "preset": "jest-preset-angular",
    "setupFilesAfterEnv": ["<rootDir>/setup-jest.ts"],
    "globalSetup": "jest-preset-angular/global-setup"
  }
}
```

Adjust your `tsconfig.spec.json` to be:

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/spec",
    "module": "CommonJs",
    "types": ["jest"]
  },
  "include": ["src/**/*.spec.ts", "src/**/*.d.ts"]
}
```

Configuration 3/3

To run our tests with Jest we can now modify the test script to use Jest instead of ng test withing the package.json.

```
"test": "jest",
```

Clean your folder

```
npm uninstall karma karma-chrome-launcher karma-coverage-istanbul-reporter karma-jasmine karma-jasmine-html-reporter
```

Adapt your test in component.spec

```
test(`the title is 'my-awesome-app'`, async(() => {
  const fixture = TestBed.createComponent(AppComponent)
  const app = fixture.debugElement.componentInstance
  expect(app.title).toEqual('my-awesome-app')
}))
```

Remove src/test.ts and karma.conf.js

```
rm src/test.ts
rm karma.conf.js
```

Remove the target "test" of the "angular.json" file

Run the test

run npm test,

```
npm test
```

Write more complex test

```
import {   ComponentFixture, fakeAsync, TestBed } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  let component: AppComponent;
  let fixture: ComponentFixture<AppComponent>;

  beforeEach(fakeAsync(() => {
      TestBed.configureTestingModule({
          imports: [RouterTestingModule],
          declarations: [ AppComponent ]
      });
      fixture = TestBed.createComponent(AppComponent);
      component = fixture.debugElement.componentInstance;
  }));

  test('should exist', () => {
      expect(component).toBeDefined();
  });
  test(`the title is 'my-awesome-app'`, fakeAsync(() => {
    expect(component.title).toEqual('my-awesome-app')
  }))
});
```

Unit-Test d’un Service 1/2

```
describe('PickyWeatherStation', () => {

    it('should give temperature', fakeAsync(() => {
        const weatherStation: PickyWeatherStation = TestBed.get(PickyWeatherStation);
        let temperature;
        weatherStation.getTemperature('Lyon')
            .subscribe(_temperature => temperature = _temperature);
        expect(temperature).toBe(42);
    }));

});
```

La méthode statique **TestBed.get** permet d’injecter les services dans les tests unitaires.

Unit-Test d’un Service 2/2

Pour éviter de récupérer l’instance dans chaque "spec", pensez à utiliser la fonction beforeEach

```
let weatherStation: PickyWeatherStation;
beforeEach(() => weatherStation = TestBed.get(PickyWeatherStation))
```

Unit-Test d’un Service 2/2

```
describe('PickyWeatherStation', () => {
    let weatherStation: PickyWeatherStation;
    beforeEach(inject([PickyWeatherStation], _weatherStation => weatherStation = _weatherStation));
    it('should give temperature', fakeAsync(() => {
        let temperature;

        weatherStation.getTemperature('Lyon')
            .subscribe(_temperature => temperature = _temperature);
        expect(temperature).toBe(42);
    }));
});
```

La fonction inject peut être utilisée directement autour de la fonction de "spec" mais il est généralement plus simple d’ajouter un beforeEach pour chaque service afin d’éviter les problèmes liés à l’ordre des tokens d’injection et le refactoring en général.

Unit-Test d’un Composant 1/2

Pour déclencher des événements sur le DOM (e.g. : changement d’un input de formulaire), il faut utiliser la méthode native dispatchEvent.

```
const input: HTMLInputElement = debugElement.nativeElement.querySelector('input');
input.value = 'test';
input.dispatchEvent(new Event('input'));
```

N’oubliez pas d’appeler la méthode detectChanges dès l’instanciation du composant pour initialiser le formulaire et permettre à Angular d’ajouter les bons listeners etc…

Unit-Test d'un Composant 2/2

```
describe('AppComponent', () => {
  let component: AppComponent;
  let fixture: ComponentFixture<AppComponent>;
  let debugElement: DebugElement;
  beforeEach(fakeAsync(() => {
      TestBed.configureTestingModule({
          imports: [RouterTestingModule],
          declarations: [ AppComponent ]
      });
      fixture = TestBed.createComponent(AppComponent);
      component = fixture.debugElement.componentInstance;
      debugElement = fixture.debugElement
      fixture.detectChanges();
  }));

  test('should exist', () => {
      expect(component).toBeDefined();
  });
  test(`the title is 'my-awesome-app'`, fakeAsync(() => {
    expect(component.title).toEqual('my-awesome-app')
  }))
});
```

Test coverage

Jest provides code coverage out of the box. Open up package.json and add a script:

```
"test:cov": "jest --coverage"
```

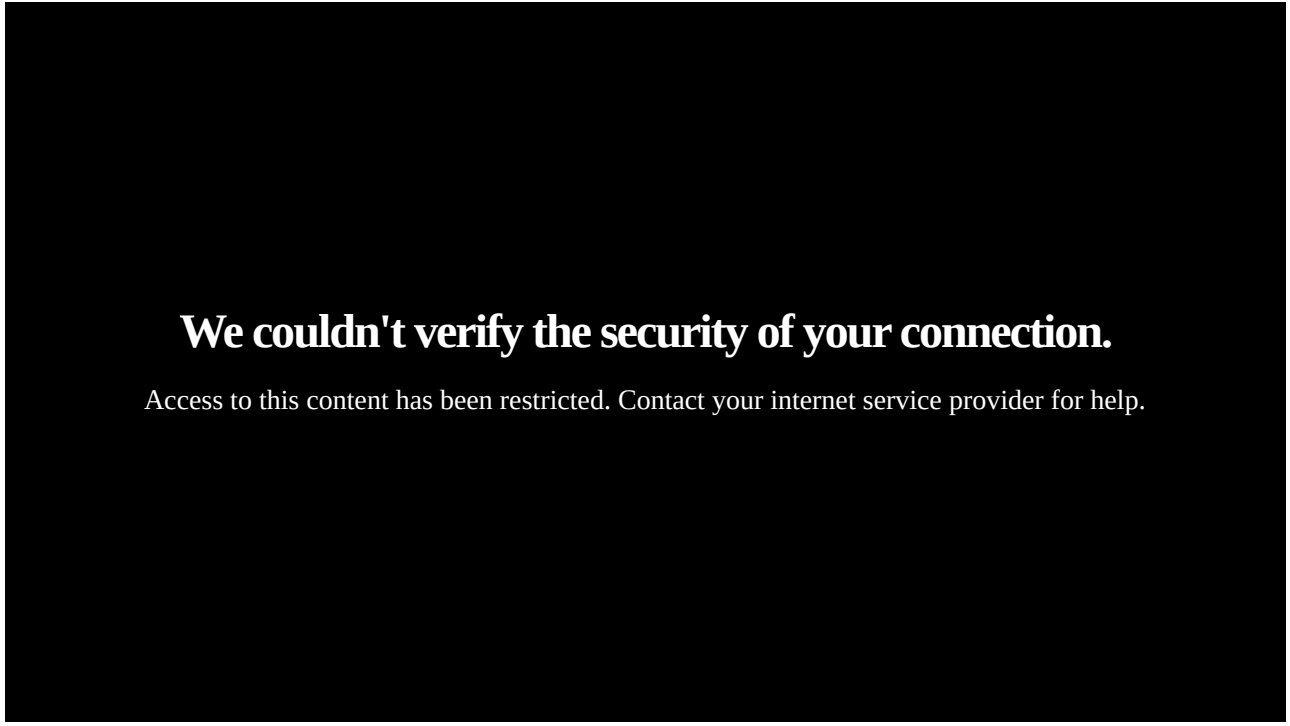
next run

```
npm run test:cov
```

Tips

- Difference between *it* and *test* => **The Jest docs state it is an alias of test.**
- Should I use both Cypress and Jest together? => It's very common to use Jest and Cypress in the same codebase. Jest for **Unit, Integration tests** and Cypress for **E2E tests**

Cypress for E2E testing



References

- https://medium.com/@nerdic.coder/how-to-use-jest-unit-tests-with-angular-87509b500158
- https://semaphoreci.com/community/tutorials/testing-angular-2-and-continuous-integration-with-jest

Questions?