



formation.tech

Tester des applications Angular



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 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 inaccessible manuellement (bouton caché par une popup...)



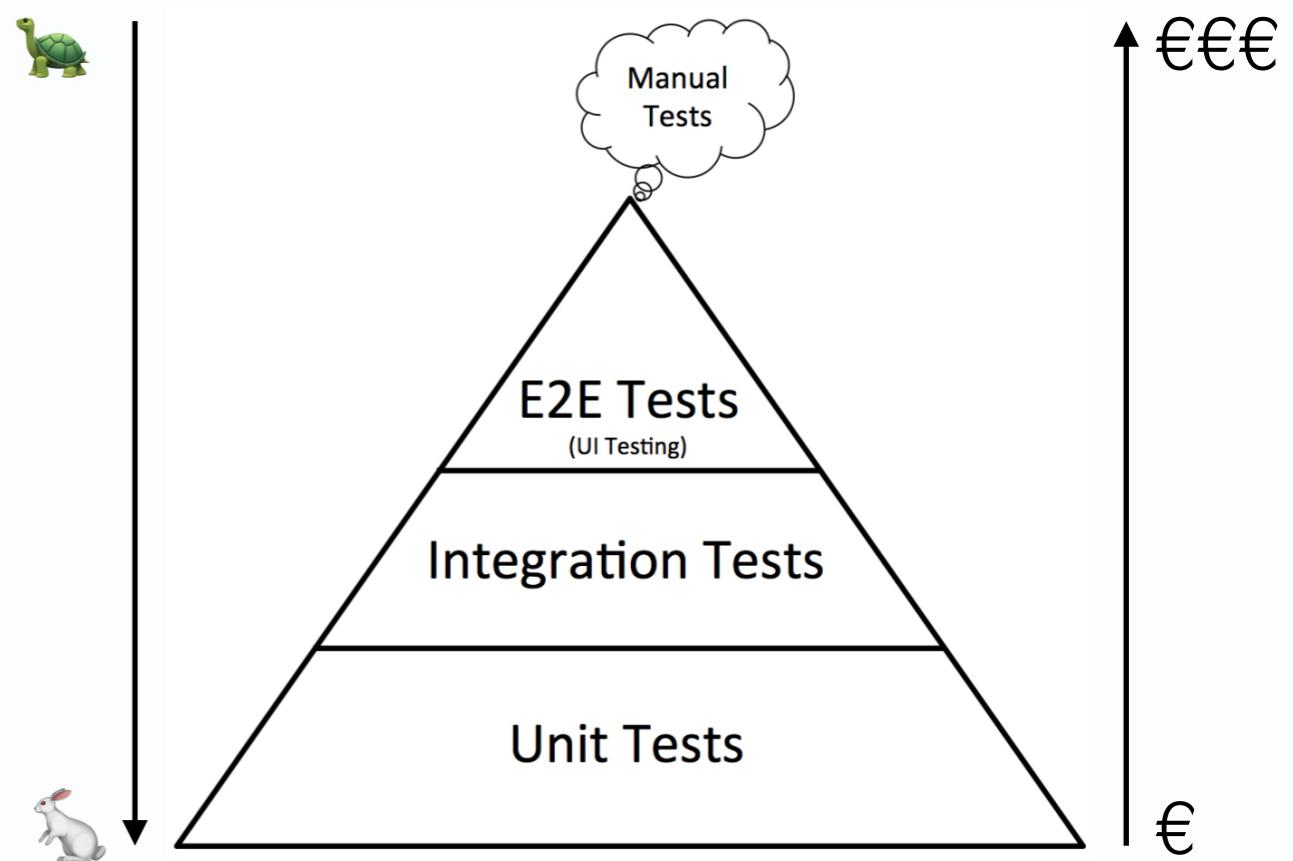
Tests Automatisés - Types de tests

- Types de tests :
 - tests de code statiques / linters
 - tests de code dynamiques / tests unitaires...
 - tests de déploiement
 - tests de sécurité
 - 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
- Un 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 continue (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 - Introduction



- Crée en 2010
- Licence MIT
- Intègre son propre système de matching et de doubles
- Utilise le style BDD

Jasmine - Mise en place



- Installation
`npm i jasmine -D`
- Initialisation
`npx jasmine init`
- Lancement des tests
`npx jasmine`
- Via le script test
`npm run-script test`
`npm run test`
`npm test`
`npm t`

Jasmine - Hello, world



- Jasmine propose 3 fonctions de base
 - describe, permet de définir une suite de tests via un callback
 - on peut imbriquer les describe pour créer des sous-groupes
 - it, permet de définir le test
 - expect, permet d'exprimer l'attente (le résultat de ce test devrait être)

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

describe('sum function', () => {
  it('should add positive number', () => {
    expect(sum(1, 2)).toEqual(3);
  });
});
```



Jasmine - Partager des variables entre les tests

- › On peut utiliser la portée de closure

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

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

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

- › Ou bien le mot clé this (ne pas utiliser les fonctions fléchées dans ce cas)

```
describe('A suite is just a function', function () {
  this.a;

  it('and so is a spec', function () {
    this.a = true;

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

Jasmine - 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;
  });
});
```

Jasmine - Désactiver les tests



- › Pour désactiver certains tests on peut :
 - les commenter
 - utiliser la fonction *xit* au lieu de *it*
 - utiliser la fonction *xdescribe* au lieu de *describe*

```
describe('sum function', () => {
  xit('should add positive number', () => {
    expect(sum(1, 2)).toEqual(3);
  });
  it('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
```

Jasmine - Désactiver les tests



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

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 dépends 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 .andRejectWith 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
 - utiliser les spies

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');
    }, 10000);
  });
}

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
- 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.spy`, `spyOn...`) pour isoler les tests de l'extérieur (API réseaux, de stockage...)



formation.tech

Karma



Karma - Introduction

- › Lanceur de test
Permet de lancer vos tests simultanément dans Chrome, Firefox, Internet Explorer...
- › Installation
 npm install karma —save-dev
 npm install -g karma-cli
- › Configuration du projet
 karma init
- › Lancement des tests
 karma start

```
Air-de-Romain:Jasmine romain$ karma init

Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no

Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
> Safari
>

What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
>
```

```
Air-de-Romain:Jasmine romain$ karma start
02 09 2015 21:30:11.510:INFO [karma]: Karma v0.13.9 server started at http://localhost:9876/
02 09 2015 21:30:11.518:INFO [launcher]: Starting browser Chrome
02 09 2015 21:30:11.526:INFO [launcher]: Starting browser Safari
02 09 2015 21:30:12.723:INFO [Safari 8.0.7 (Mac OS X 10.10.4)]: Connected on socket HE38s1HTBKXL5t5yAAAA with id 54715269
Safari 8.0.7 (Mac OS X 10.10.4): Executed 1 of 1 SUCCESS (0.038 secs / 0.003 secs)
Safari 8.0.7 (Mac OS X 10.10.4): Executed 1 of 1 SUCCESS (0.038 secs / 0.003 secs)
Chrome 45.0.2454 (Mac OS X 10.10.4): Executed 1 of 1 SUCCESS (0.04 secs / 0.008 secs)
TOTAL: 2 SUCCESS
```



Karma - Plugins

- On peut installer des plugins supplémentaires pour Karma
<http://karma-runner.github.io/latest/dev/plugins.html>
- 4 types de plugins
 - Frameworks : choix du framework (jasmine, mocha...)
 - Reporters : rapports (JUnit, Notifications Systèmes, Istanbul...)
 - Launchers : plateforme où lancer les tests (navigateurs, SauceLabs/ BrowserStack...)
 - Preprocessors (TypeScript...)

Karma - Config



- La config se définit dans le fichier karma.conf.js

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular-devkit/build-angular'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage-istanbul-reporter'),
      require('@angular-devkit/build-angular/plugins/karma')
    ],
    client: {
      clearContext: false // leave Jasmine Spec Runner output visible in browser
    },
    coverageIstanbulReporter: {
      dir: require('path').join(__dirname, './coverage/app-angular'),
      reports: ['html', 'lcovonly', 'text-summary'],
      fixWebpackSourcePaths: true
    },
    reporters: ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    LogLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false,
    restartOnFileChange: true
  });
};
```



formation.tech

Tests Unitaires sous Angular



Tests Unitaires sous Angular - Introduction

- Par défaut, les tests unitaires et d'intégration sont écrits avec Jasmine et se lancent via Karma
- D'autres choix existent, par exemple pour Jest :
<https://github.com/just-jeb/angular-builders>
- On les exécute avec la commande :
`ng test`
- Le module `@angular/core/testing` fournit un certain nombre d'outils de test
- Certains modules liés à des libs également, par exemple
 - `@angular/common/http/testing`
 - `@angular/router/testing`



Tests Unitaires sous Angular - Code Coverage

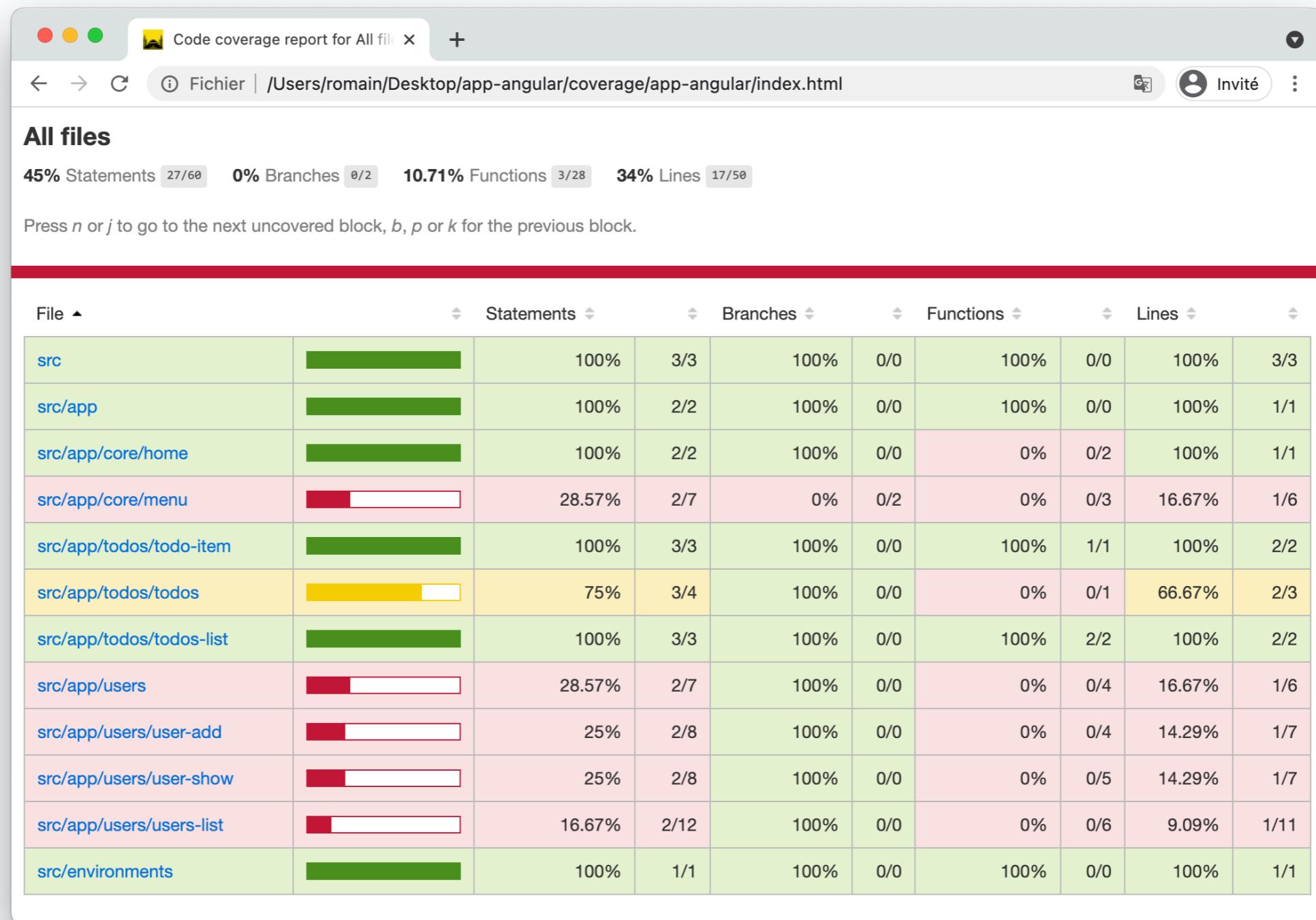
- › Pour calculer la couverture de code (% de code exécuté lors de test), on utilise la commande :
`ng test --code-coverage`
- › Dans la console

```
===== Coverage summary =====
Statements    : 45% ( 27/60 )
Branches     : 0% ( 0/2 )
Functions    : 10.71% ( 3/28 )
Lines        : 34% ( 17/50 )
=====
```



Tests Unitaires sous Angular - Code Coverage

- Un rapport HTML est créé dans le dossier coverage





Tests Unitaires sous Angular - Code Coverage

- On peut voir les lignes qui sont testée ou non par fichier

The screenshot shows a code coverage report for the file `todos.component.ts`. The report indicates the following coverage metrics:

- Statements: 75% (3/4)
- Branches: 100% (0/0)
- Functions: 0% (0/1)
- Lines: 66.67% (2/3)

A message at the top of the code editor says: "Press `n` or `j` to go to the next uncovered block, `b`, `p` or `k` for the previous block."

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-todos',
5   templateUrl: './todos.component.html',
6   styleUrls: ['./todos.component.scss']
7 })
8 1x export class TodosComponent {
9
10 1x   public todos = ['Item 1', 'Item 2', 'Item 3'];
11
12   public handleNewTodo(todo) {
13     this.todos = [todo, ...this.todos];
14   }
15
16 }
17
```

Code coverage generated by [istanbul](#) at Mon May 24 2021 22:55:44 GMT+0200 (Central European Summer Time)



Tests Automatisés - Test de composant

- Structure d'un test

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { HelloComponent } from './hello.component';

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

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ HelloComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(HelloComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```



Tests Automatisés - Test de composant

- Le premier beforeEach configure un module spécifique au test

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [ HelloComponent ]
  })
  .compileComponents();
});
```

- Tous les imports, providers, declarations pourront être adapté pour le test du composant
- Mais il va falloir configurer une nouvelle fois le module dans chaque test
- La fonction est asynchrone car le compilateur l'est



Tests Automatisés - Test de composant

- Le second beforeEach est synchrone

```
beforeEach(() => {
  fixture = TestBed.createComponent(HelloComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

- Il va instancier le composant et l'afficher dans le DOM
- Si on passe des valeurs manuellement il faudra lancer la détection de changement soi même



Tests Automatisés - Test de composant

- Le test basique vérifie que le composant peut être créé

```
it('should create', () => {
  expect(component).toBeTruthy();
});
```

- Si le module de test est mal configuré, le test va échouer.



Tests Automatisés - Test de composant

- Tester le contenu du composant

```
<p>hello works!</p>
```

- Pour sélectionner des éléments dans un test on peut utiliser
 - L'API DebugElement d'Angular

```
it('should contains hello works (debugElement)', () => {
  const p = fixture.debugElement.query(By.css('p'))
  expect(p.nativeElement.textContent).toBe('hello works');
});
```

- Le DOM

```
it('should contains hello works (DOM)', () => {
  const p = fixture.nativeElement.querySelector('p');
  expect(p.textContent).toBe('hello works');
});
```



Tests Automatisés - Test de composant

- › Bonne pratique : ne pas être trop précis pour simplifier la maintenance

```
it('should contains hello works (bonne pratique)', () => {
  expect(fixture.nativeElement.textContent).toContain('hello works');
});
```

- › Il n'était pas nécessaire de sélectionner le paragraphe pour que le test passe
- › Privilégier des méthodes comme *.toContain* à *.toBe*



Tests Automatisés - Test de composant

- Tester les @Input

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-hello',
  template: '<p>hello {{name}}!</p>'
})
export class HelloComponent {
  name = 'Romain';
}
```

- Ce test passe

```
it('should contains hello Romain!', () => {
  expect(fixture.nativeElement.textContent).toContain('hello Romain!');
});
```

- Mais pas celui là

```
it('should contains hello Romain!', () => {
  component.name = 'Jean'
  expect(fixture.nativeElement.textContent).toContain('hello Jean!');
});
```



Tests Automatisés - Test de composant

- Rappel sur la détection de changement, 3 cas :
 - utilisation de Event Binding dans un template
 - utilisation de callbacks asynchrones grâce à Zone.js
 - utilisation du service ChangeDetectorRef
- Donc programmatiquement la ligne suivante ne lance pas la détection de changement

```
component.name = 'Jean';
```

- Il faudra la lancer manuellement

```
it('should contains hello Romain!', () => {
  component.name = 'Jean';
  fixture.detectChanges();
  expect(fixture.nativeElement.textContent).toContain('hello Jean!');
});
```



Tests Automatisés - Test de composant

- Tester des événements

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-counter',
  template: '<button (click)="handleClick()">{{ count }}</button>'
})
export class CounterComponent {
  count = 0;

  handleClick() {
    this.count++;
  }
}
```

- Avec l'API DebugElement d'Angular

```
it('should increment on click (DebugElement)', () => {
  const button = fixture.debugElement.query(By.css('button'));
  button.triggerEventHandler('click', {});
  expect(component.count).toBe(1);
});
```

- Avec le DOM

```
it('should increment on click (DOM)', () => {
  const button = fixture.nativeElement.querySelector('button');
  button.dispatchEvent(new MouseEvent('click'));
  expect(component.count).toBe(1);
});
```



Tests Automatisés - Test de composant

- Tester du code qui dépend d'un appel HTTP

```
import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { Observable, of } from 'rxjs';

@Component({
  selector: 'app-random-user',
  template: '<p>{{ (user$ | async)?.name }}</p>'
})
export class RandomUserComponent implements OnInit {
  user$: Observable<any> = of();

  constructor(private httpClient: HttpClient) { }

  ngOnInit(): void {
    this.user$ = this.httpClient.get('https://jsonplaceholder.typicode.com/users/1')
  }
}
```



Tests Automatisés - Test de composant

- Dans le module de test, veiller à bien importer les modules, par exemple HttpClientModule

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [ RandomUserComponent ],
    imports: [HttpClientModule],
  })
  .compileComponents();
});
```

- Le test :

```
it('should show user name', async () => {
  await fixture.whenStable();
  fixture.detectChanges();
  expect(fixture.nativeElement.textContent).toContain('Leanne Graham');
});
```

- fixture.whenStable va s'exécuter lorsque tous les callbacks asynchrones auront été intercepté par Zone.js après avoir été exécutés



Tests Automatisés - Test de composant

- L'idéal serait de ne pas exécuter la requête pour ne pas dépendre de l'API REST
- 3 approches :
 - les spies de Jasmine
 - configurer l'injection de dépendance différemment
 - utiliser HttpClientTestingModule



Tests Automatisés - Test de composant

- Avec les spies de Jasmine

```
beforeEach(() => {
  const httpClient = TestBed.inject(HttpClient);
  spyOn(httpClient, 'get').and.returnValue(of({name: 'Toto'}))

  fixture = TestBed.createComponent(RandomUserComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});

it('should show user name', () => {
  expect(fixture.nativeElement.textContent).toContain('Toto');
});
```



Tests Automatisés - Test de composant

- En configurant l'injection de dépendance autrement

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [RandomUserComponent],
    providers: [
      { provide: HttpClient,
        useValue: {
          get() { return of({ name: 'Toto' }); },
        },
      },
    ],
  }).compileComponents();
});

it('should show user name', () => {
  expect(fixture.nativeElement.textContent).toContain('Toto');
});
```



Tests Automatisés - Test de composant

- En utilisant HttpClientTestingModule

```
let httpTestingController: HttpTestingController;

beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [RandomUserComponent],
    imports: [HttpClientTestingModule],
  }).compileComponents();
});

beforeEach(() => {
  httpTestingController = TestBed.inject(HttpTestingController);
  fixture = TestBed.createComponent(RandomUserComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});

it('should show user name', () => {
  const req = httpTestingController.expectOne('https://jsonplaceholder.typicode.com/users/1');

  expect(req.request.method).toBe('GET')
  req.flush({
    name: 'Toto',
  });

  fixture.detectChanges();
  expect(fixture.nativeElement.textContent).toContain('Toto');
  httpTestingController.verify();
});
```



Tests Automatisés - Voir aussi

- Bibliothèque de double spécifique à Angular
<https://ng-mocks.sudo.eu/>
- Utilitaires autour des tests Angular
<https://ngneat.github.io/spectator/docs/testing-components>
<https://testing-library.com/docs/angular-testing-library/intro/>



formation.tech

Tests E2E



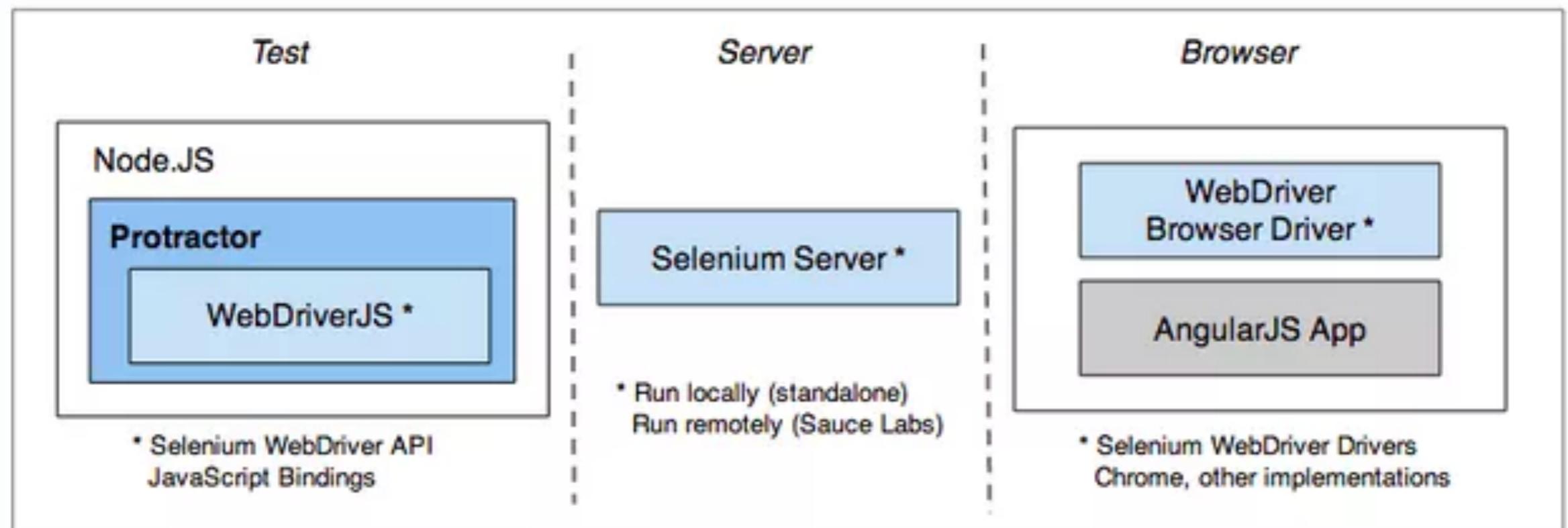
Tests E2E - Introduction

- Les tests E2E utilisent Protractor (surchouche de Selenium) et Jasmine
- On les exécute avec la commande :
`ng e2e`
- Depuis Angular 12, Protractor n'est plus installé par défaut pour laisser le choix d'utiliser d'autres bibliothèques de test, par exemple
 - Cypress
 - Webdriver.io
 - CodeceptJS
 - CucumberJS
 - Puppeteer
 - ...



Tests E2E - Prérequis

- Protractor nécessite l'utilisation de WebDriverJS, c'est à dire la version de Selenium normée par le W3C
- Il faudra installer des drivers pour exécuter dans les navigateurs





Tests E2E - Page Object

- Lors de l'écriture de Test E2E on privilégiera l'utilisation du Pattern Page Object
- Pour cela nous allons créer une classe contenant des méthodes pour interagir avec une certaine page, par exemple pour un formulaire de login
 - navigateToLoginPage
 - fillUsername
 - fillPassword
 - clickOnSubmitButton
 - getErrorsDiv
 - ...



Tests E2E - Page Object

- Exemple de Page Object

```
import { browser, by, element } from 'protractor';

export class AppPage {
  navigateTo(): Promise<unknown> {
    return browser.get(browser.baseUrl) as Promise<unknown>;
  }

  getTitleText(): Promise<string> {
    return element(by.css('app-root .content span')).getText() as Promise<string>;
  }
}
```



Tests E2E - Exemple de test

- Protractor utilise Jasmine comme framework de test
- On peut ainsi utiliser les fonctions describe, it, beforeEach, afterEach, expect...

```
import { browser, logging } from 'protractor';

import { AppPage } from './app.po';

describe('workspace-project App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  afterEach(async () => {
    // Assert that there are no errors emitted from the browser
    const logs = await browser.manage().logs().get(logging.Type.BROWSER);
    expect(logs).not.toContain(jasmine.objectContaining({
      level: logging.Level.SEVERE,
    } as logging.Entry));
  });
});
```