

Test unit JS

Typologie des tests

- ▶ Tests unitaires
 - ▶ Les tests unitaires consistent à tester individuellement les composants de l'application. On pourra ainsi valider la qualité du code et les performances d'un module.
- ▶ Tests d'intégration
 - ▶ Ces tests sont exécutés pour valider l'intégration des différents modules entre eux et dans leur environnement exploitation définitif. Ils permettront de mettre en évidence des problèmes d'interfaces entre différents programmes.
- ▶ Tests fonctionnels
 - ▶ Ces tests ont pour but de vérifier la conformité de l'application développée avec le cahier des charges initial. Ils sont donc basés sur les spécifications fonctionnelles et techniques.
- ▶ Tests de non-régression
 - ▶ Les tests de non-régression permettent de vérifier que des modifications n'ont pas altérées le fonctionnement de l'application. L'utilisation d'outils de tests, dans ce domaine, permet de faciliter la mise en place de ce type de tests.

Typologie des tests

▶ Tests IHM

- ▶ Les tests IHM ont pour but de vérifier que la charte graphique a été respectée tout au long du développement pour contrôler :
 - ▶ la présentation visuelle : les menus, les paramètres d'affichages, les propriétés des fenêtres, les barres d'icônes, la résolution des écrans, les effets de bord,...
 - ▶ la navigation : les moyens de navigations, les raccourcis, le résultat d'un déplacement dans un écran,...

▶ Tests de configuration

- ▶ Une application doit pouvoir s'adapter au renouvellement de plus en plus fréquent des ordinateurs. Il s'avère donc indispensable d'étudier l'impact des environnements d'exploitation sur son fonctionnement.

Typologie des tests

► Tests de performance

- Le but principal des tests de performance est de valider la capacité qu'ont les serveurs et les réseaux à supporter des charges d'accès importantes.
- On doit notamment vérifier que les temps de réponse restent raisonnables lorsqu'un nombre important d'utilisateurs sont simultanément connectés à la base de données de l'application.
- Pour cela, il faut d'abord relever les temps de réponse en utilisation normale, puis les comparer aux résultats obtenus dans des conditions extrêmes d'utilisation.
- Une solution est de simuler un nombre important d'utilisateur en exécutant l'application à partir d'un même poste et en analysant le trafic généré.
- Le deuxième objectif de ces tests est de valider le comportement de l'application, toujours dans des conditions extrêmes. Ces tests doivent permettre de définir un environnement matériel minimum pour que l'application fonctionne correctement.

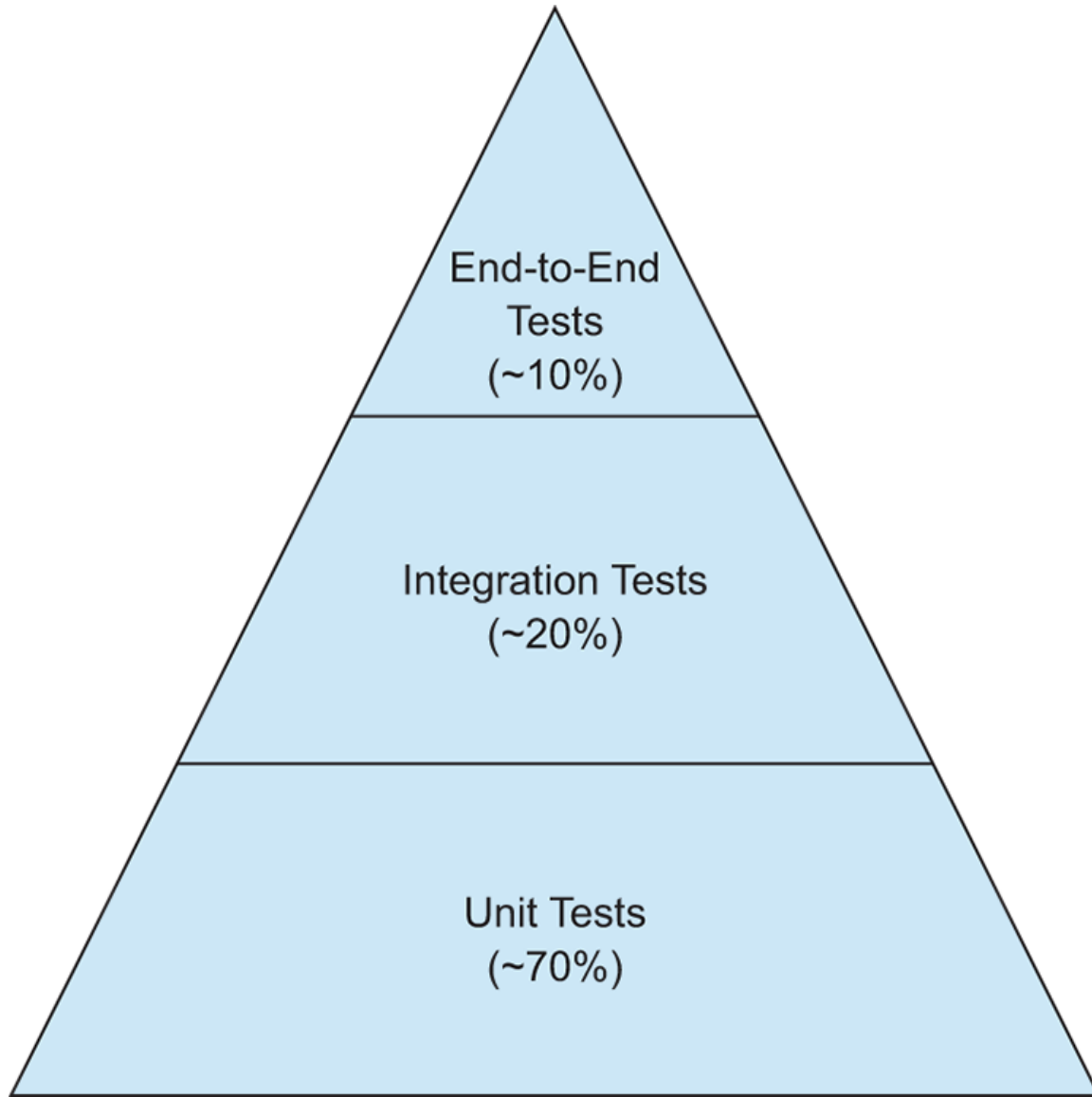
► Tests d'installation

- Une fois l'application validée, il est nécessaire de contrôler les aspects liés à la documentation et à l'installation.
- Les procédures d'installation doivent être testées intégralement car elles garantissent la fiabilité de l'application dans la phase de démarrage.
- Bien sûr, il faudra aussi vérifier que les supports d'installation ne contiennent pas de virus.

Test end-to-end

- ▶ Tester l'application comme dans le vrai monde avec un vrai utilisateur simulé.
- ▶ Ces tests lancent un navigateur, navigue dans l'application et interagit avec.
- ▶ Chonophage donc bien choisir les scénarios.

Pyramide des tests



- ▶ Recommandation pour bien tester une application
- ▶ Dans les tests d'intégration, on va tester l'appel aux données (API, DB, etc...) qui sera *mocké* dans les test e2e
- ▶ Les test unitaires couvrent beaucoup de parties de l'application, ce qui allège les tests e2e

Frameworks de tests (ex: Angular)

- ▶ Lancer les tests : `ng test`
- ▶ Karma
 - ▶ Moteur de tests : crée une instance d'un navigateur, run les tests et fournit les résultats.
Avantages : en ligne de commande, il refresh le navigateur automatiquement.
 - ▶ `npm install karma karma-chrome-launcher karma-jasmine karma-cli`
- ▶ Jasmine
 - ▶ Test unitaire
- ▶ Selenium
 - ▶ Test e2e
- ▶ Protractor
 - ▶ Test e2e
- ▶ Jtest
 - ▶ Test unitaire

Karma par l'exemple

- Nouvelle application avec un composant Pizza créé automatiquement

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

@Component({
  selector: 'app-pizza',
  templateUrl: './pizza.component.html',
  styleUrls: ['./pizza.component.css']
})

export class PizzaComponent implements OnInit {

  title = "I love pizza!"

  constructor() { }

  ngOnInit() {
  }

}
```

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { PizzaComponent } from './pizza.component';

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

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

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


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


Karma par l'exemple

- Il suffit de rajouter dans `pizza.component.spec.ts` :
- ```
it(`should have a title 'I love pizza!'`, async(() => {
 fixture = TestBed.createComponent(PizzaComponent);
 component = fixture.debugElement.componentInstance;
 expect(component.title).toEqual('I love pizza!');
}));
```

**Karma v4.1.0 - connected** DEBUG

Chrome 79.0.3945 (Mac OS X 10.15.2) is idle

 **Jasmine** 3.4.0 Options

● ● ● ● ●

5 specs, 0 failures, randomized with seed 51455 finished in 0.717s

AppComponent

- should render title
- should create the app
- should have as title 'ng-unit-test'

PizzaComponent

- should have as title 'I love pizza!'
- should create

# Test Jasmine

- ▶ Fichier \*-spec.ts
- ▶ 1 fichier = 1 describe
- ▶ 1 it = 1 test
- ▶ Le test effectif est `expect(ceQuiDoitEtreTesté).toXXX(resultAttendu);`
- ▶ On peut tester l'égalité, le fait d'être non null, d'être null, la supérior, etc...
  - ▶ => <https://jasmine.github.io/api/3.6/matchers.html>

```
describe("Hello World", function() {

 it("should Return Hello world",function() {
 expect(helloworld()).toEqual('Hello World');
 });

});
```

```
describe("My first suite", function() {
 it("contains spec with an expectation", function() {
 expect(true).toBe(true);
 });
});
```

# Before/after

- ▶ Il est possible d'exécuter du code « avant » ou « après » chacune des *specs* écrites, respectivement grâce aux fonctionnalités **beforeEach** et **afterEach**.
- ▶ Cela peut devenir très pratique si l'on veut factoriser du code, ou si l'on utilise des variables globales que l'on souhaite réinitialiser après un test.

```
describe("My first suite with 'beforeEach' and 'afterEach'", function() {
 var a = 0;

 beforeEach(function() {
 a += 1;
 });

 afterEach(function() {
 a = 0;
 });

 it("checks the value of a", function() {
 expect(a).toEqual(1);
 a += 1;
 });

 it("expects a to still be equal to 1", function() {
 expect(a).toEqual(1);
 });
});
```

# Before/after

- ▶ Il est également possible d'exécuter du code « avant » ou « après » toutes les *specs* contenues dans une suite.
- ▶ Comme le suggère son nom, la fonction **beforeAll** est appelée avant l'exécution de toutes les *specs*, et **afterAll** est appelée après l'exécution de toutes les *specs*.

```
describe("My first suite with 'beforeAll' and 'afterAll'", function() {
 var a;

 beforeAll(function() {
 a += 1;
 });

 afterAll(function() {
 a = 0;
 });

 it("checks the value of a", function() {
 expect(a).toEqual(1);
 a += 1;
 });

 it("does not reset a between specs", function() {
 expect(a).toEqual(2);
 });
});
```

# Machters

```
describe("My first suite", function() {
 var a = true;

 it("contains spec with an expectation", function() {
 expect(a).not.toBe(false);
 });
});
```

- ▶ Un *matcher* produit une **comparaison booléenne** entre la valeur réelle d'un élément et sa valeur attendue. Si ces deux valeurs sont divergentes, l'*expectation* est considérée comme fausse, et Jasmine fait échouer la *spec*.
- ▶ Un *matcher* peut aussi être évalué avec une assertion négative. Il suffit pour cela d'ajouter le mot-clé *not* avant l'utilisation du *matcher*.
- ▶ **toBe, toEqual, toMatch, toBeNull, toContain, toBeTruthy, toBeFalsy, toBeDefined, toBeUndefined, toBeGreaterThan, toBeLessThan, toBeCloseTo, toThrow, toThrowError**

# Exemple de test

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

describe('adding a new contact with only a name', () => {
 beforeEach(() => {
 browser.get('/#/');
 });

 it('should find the add contact button', () => {
 element(by.id('add-contact')).click();
 expect(browser.getCurrentUrl())
 .toEqual(browser.baseUrl + '/#/add');
 });

 it('should write a name', () => {
 let contactName = element(by.id('contact-name'));
 contactName.sendKeys('Ada');
 expect(contactName.getAttribute('value'))
 .toEqual('Ada');
 });

 it('should click the create button', () => {
 element(by.css('.create-button')).click();
 expect(browser.getCurrentUrl())
 .toEqual(browser.baseUrl + '/#/');
 });
});
```

**Finds the name input  
field with the 'contact  
name' id attribute**

**Finds the Create button with  
the 'create-button' css class**