12/12/2022
Version 3

ANGULAR

JÉRÉMY PERROUAULT

TESTS

TESTS UNITAIRES

TESTS

- Principe tests unitaires
  - Simples, rapides, autonomes
  - « Given », « When », « Then »
    - On donne un contexte … et lorsque … on vérifie que …

TESTS

- Jasmine et Karma sont utilisés par Angular comme bibliothèque de test unitaire
  - Jasmine fourni un ensemble de fonctionnalités pour tester unitairement
  - Karma permet d'exécuter tous les tests, et de surveiller les fichiers pour les relancer
    - Dans un navigateur, il affiche le résultat des tests

- Comme vu précédemment, lorsqu'Angular CLI génère des fichiers, il génère aussi des SPEC.TS
  - Qui sont les fichiers de tests par convention

TESTS

- Pour exécuter Karma

```
ng test
ng t
```

TESTS

- Dans Jasmine, les fonctions
  - `describe` — Ensemble de tests « it »
  - `it` — Un test unitaire
  - `beforeEach / afterEach` — S'exécute avant / après chaque test unitaire
  - `beforeAll / afterAll` — S'exécute avant / après tous les tests unitaires
  - `expect` — S'attend à une Assertion particulière
    - `expect(..).toBeNull()` — S'attend à un null
    - `expect(..).not.toBeNull()` — S'attend à un non-null

TESTS

- Exemple

```
describe('DemoService', () => {
  beforeEach(() => {
    console.log("Avant chaque test.");
  });

  it('should return demo string', () => {
    const service = new DemoService();
    const result = service.demoTest();

    expect(result).toEqual('demo');
  });
});
```

EXERCICE

- Créer un tout nouveau projet
- Créer une classe **Todo** (ou recopier)
- Créer un nouveau service **TodoService**
  - Une méthode `findAll` qui retourne un tableau de **Todo**

- Tester que le service
  - Retourne bien un tableau, non null, rempli d'au moins un **Todo**
  - Retourne bien un tableau de **Todo** dont l'id et le title sont présents

EXERCICE

- Modifier le service **TodoService**
  - `findAll` retourne un **Observable**

- Tester que le service
  - Retourne bien un tableau, non null, rempli d'au moins un **Todo**
  - Retourne bien un tableau de **Todo** dont l'id et le title sont présents

TESTS

- Dans Jasmine
  - `fdescribe / fit` Permet de se **f**ocaliser sur un ensemble ou un test
  - `xdescribe / xit` Permet d'e**x**clure un ensemble ou un test

TESTS

- Dans Jasmine, les fonctions
  - `spyOn` Crée un espion, permet de retourner des valeurs fictives
  - `toHaveBeenCalled..` Vérifie l'appel à une méthode et plus (arguments, X fois, …)

```
beforeEach(() => {
  spyOn(instance, 'nomMethode')
    .and
    .returnValue(value);
});
```

```
it('should method called', () => {
  instance.nomMethode(1);

  expect(instance.nomMethode).toHaveBeenCalled();
  expect(instance.nomMethode).toHaveBeenCalledWith(1);
}
```

TESTS

- Avec la fonction `configureTestingModule`, on peut préciser des options
  - Injecter des modules, des providers, etc.
- Pour injecter **HttpClient**, il faut utiliser le module de test **HttpClientTestingModule**
  - Présent dans @angular/common/http/testing
  - Permettra de « mocker » les connexions HTTP

```
TestBed.configureTestingModule({
  imports: [ HttpClientTestingModule ]
});
```

- Avec la fonction `inject`, on peut récupérer une instance

```
http = TestBed.inject(HttpClient);
```

EXERCICE

- Modifier le service **TodoService**
  - Le service injecte **HttpClient**
  - `findAll` retourne un **Observable** via la méthode `get` de **HttpClient**

- Tester que le service
  - Retourne bien un tableau, non null, rempli d'au moins un **Todo**
  - Retourne bien un tableau de **Todo** dont l'id et le title sont présents
  - Utilise bien la méthode `get` de **HttpClient**, avec la bonne URL

TESTS

- Pour tester un composant
  - Il faut un emballage de composant / template (qu'on appelle *fixture*) qui réagira comme un navigateur
  - La fonction `createComponent` créera la *fixture* pour le composant

```
fixture = TestBed.createComponent(HelloWorldComponent);
component = fixture.componentInstance;
```

- `detectChanges` permet de lancer manuellemenet `ngOnChanges` du cycle de vie
  - Car dans ce contexte de test, ça ne sera pas automatique

```
fixture.detectChanges();
```

TESTS

- On peut manipuler la *fixture* pour récupérer l'élément HTML généré
  - Et le manipuler comme s'il était dans un navigateur

```
it('should have paragraph with text', () => {
  const element = fixture.nativeElement;

  expect(element.querySelector('p').innerText).toContain('un mot');
});
```

EXERCICE

- Créer un composant **HelloWorld**
  - Ajouter un h1 avec le texte « Hello World »

- Tester que le composant
  - Possède bien un h1 avec le contenu « Hello World »
  - Sans `detectChanges`

EXERCICE

- Modifier le composant **HelloWorld**
  - Ajouter un attribut « title » avec la valeur « Hello World »
  - Le h1 possède la valeur de l'attribut

- Tester que le composant
  - Possède bien un h1 avec le contenu « Hello World »
  - Sans `detectChanges`
  - Puis avec `detectChanges`

EXERCICE

- Modifier le composant **HelloWorld**
  - Ajouter un bouton
  - Lors du clique, modifier la valeur de *title*

- Tester que le composant, et
  - Vérifier que le `h1` a le contenu « Hello World »
  - Simuler le click sur le bouton, avec la méthode `click` de l'élément HTML
  - Vérifier que le `h1` a le nouveau contenu

TESTS

- Lorsqu'un composant a besoin d'une dépendance, un service par exemple
  - On peut le lui fournir via *providers*
    - Mais dans ce cas, on lui fourni le vrai service
  - On peut lui fournir un « fake » via une classe ou via une valeur (`useClass` / `useValue`)

```
const fakeService = {
  // TODO méthodes etc.
};
```

```
providers: [
  { provide: TodoService, useValue: fakeService }
]
```

EXERCICE

- Modifier le composant **HelloWorld**
  - Attend le service **TodoService**
  - Utilise le service pour afficher la liste des **Todo**

- Tester que le composant
  - Possède autant de **Todo** dans son Template qu'il y en a retourné par le service `findAll`

TESTS

- En utilisant ngMocks (qu'il faut installer en tant que dépendance « dev »)

```
npm install -D ng-mocks
```

- On peut récupérer une instance mockée d'un service

```
const fakeService = MockService(MonService);
```

- Utile lorsque l'on veut espionner des méthodes par exemple

```
spyOn(fakeService, 'nomMethode')
  .and
  .returnValue(value);
```

EXERCICE

- Modifier le test du composant pour vérifier que `findAll` du service est bien appelé

TESTS E2E

TESTS DE BOUT EN BOUT

TESTS E2E

- Les tests de bout en bout (end-to-end ou e2e)
  - Permet de tester l'application dans un vrai contexte en simulant les actions de l'utilisateur
  - Sont plus lents que les tests unitaires

TESTS E2E

- Protractor était utilisé pour exécuter les tests dans un contexte d'application
  - Déprécié depuis mai 2021
  - Cypress, Nightwatch ou WebdriverIO sont proposés par Angular en remplacement

- Protractor utilisait Jasmine pour décrire les tests
- Cypress utilise Mocha (et non Jasmine) pour décrire les tests, et Chai pour les Assertions
  - Cypress surveille les fichiers pour relancer les tests

TESTS E2E

- Pour exécuter Cypress, Nightwatch ou WebdriverIO

```
ng e2e
ng e
```

  - Si l'environnement pour les tests e2e n'a pas été trouvé
    - Angular CLI propose d'ajouter la bibliothèque en proposant les 3 solutions

  - Contrairement aux tests unitaires, les fichiers de tests ne sont pas créés
    - Pour Cypress, les fichiers seront CY.TS

TESTS E2E

- Pour faire exécuter tous les tests à Cypress

```
ng serve
```

```
npm run cypress:run
```

TESTS E2E

- Exemple

```
describe('HomePage', () => {
  it('Should text and h2 exist', () => {
    cy.visit('/');
    cy.contains('un mot');
    cy.get('h2').should('exist’);
    cy.get('h2').should('text', 'Le mot');
    cy.get('h2').should('not.text', 'Le pas mot’);
    cy.get('button').click();
  });
});
```

https://docs.cypress.io/

EXERCICE

- Ajouter Cypress et lancer le premier test par défaut

```
ng e2e
```

EXERCICE

- Tester la page d'accueil
  - Le contenu du `h1` doit être « Hello World »
  - Le clique sur le bouton (méthode `click()`)
  - Le contenu du `h1` doit avoir changé
  - Le nombre de `li` doit être supérieur à 2

```
..should('have.length.of.at.most', 2)
```