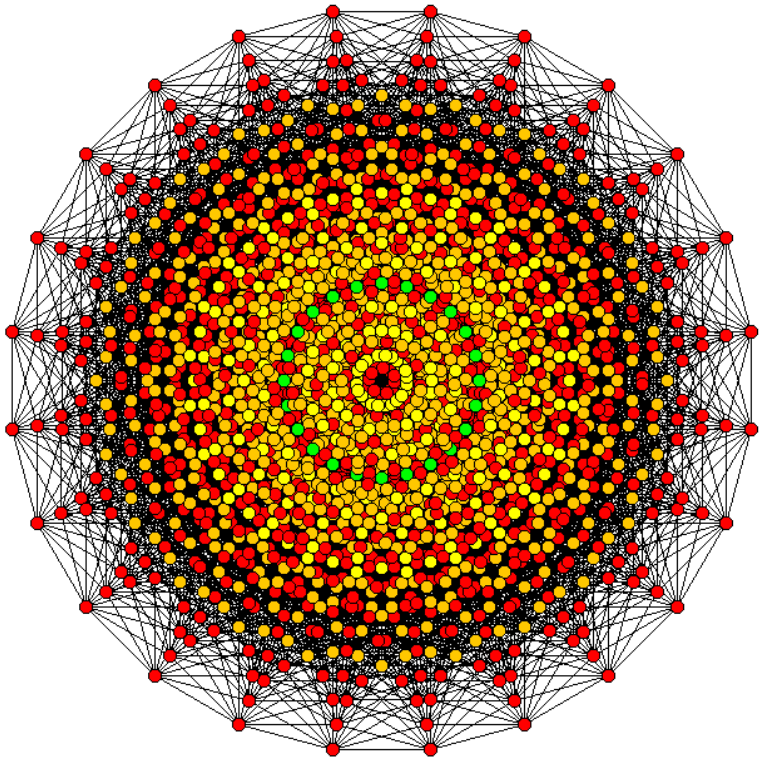


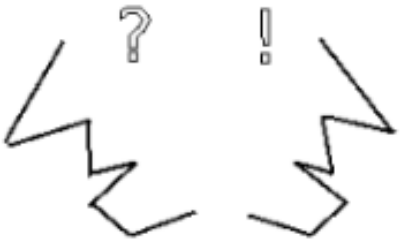
Test unitaire

O. Capuozzo

Version 1.0, 2020-11-25

Table des matieres

Présentation



Support de cours sur l'initiation aux tests unitaires

Prise en main de la gestion des tests avec Angular

Prérequis

- Des bases en programmation objet
- Avoir pris en main le framework Angular à travers un tutoriel et avoir réalisé des exercicies en autonomie
- Avoir une première expérience dans les tests unitaires

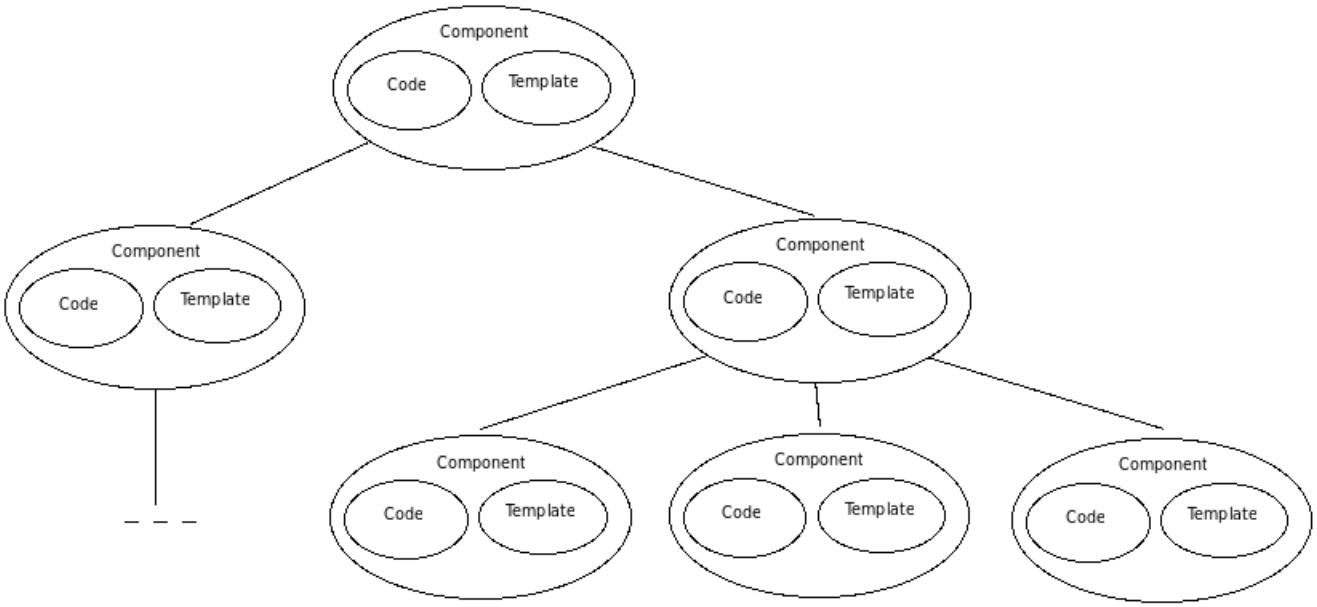
Contexe Angular

La programmation d'application frontend avec Angular (mais pas seulement) est basée sur la notion de *Component* (composant). Les composants pouvant être regroupés en *ngModules* (un package à la Angular).

Un composant regroupe :

- une structure HTML pour son rendu dans la page où il est déclaré
- une logique de comportement (actions, événements) définit par une classe Typescript
- un sélecteur CSS qui définit comme utiliser le composant dans un template parent
- optionnellement, des styles CSS à appliquer au rendu de ce composant

Pour synthétiser, une application frontent basée sur JS/HTML/CSS et le concept de Composant forme un arbre conceptuel de cette forme.



Tester un composant

Le test d'un composant consiste à vérifier si ce dernier se comporte comme attendu.

Le composant testé est appelé *component-under-test* (CUT).

- Le rendu est-il correct ?
- Le comportement du composant, le code, est-il fidèle à nos attentes, à ses spécifications ?

On comprend alors que l'on ne peut tester un tel composant sans un contexte d'environnement.

Au minimum, un contexte de composant est composé de :

- un template "parent" pour le rendu du composant
- une instance de la classe TypeScript liée au composant.

Ces deux problèmes ont leur solution dans l'écosytème Angular.

- Pour interpréter le rendu HTML du composant, on peut soit simuler le comportement d'un navigateur, soit faire directement appel à un navigateur.
- Il est courant que l'instance de la classe TypeScript du composant utilise elle-même des services qui lui sont injectés, ce qui rend plus difficile l'instanciation de cette classe.

> Un `CUT` n'a pas besoin d'être injecté avec de vrais services. En fait, il est généralement préférable d'injecter des sortes de "doublures" (*stubs*, *fakes*, *spies*, ou *mocks*). Le but de la spécification est de tester le composant, pas le service, et les services réels peuvent poser problème.
>
> Injecter le vrai service pourrait être un cauchemar. Le service réel peut demander à l'utilisateur des informations de connexion et tenter d'accéder à un serveur d'authentification. Ces comportements peuvent être difficiles à intercepter. Il est beaucoup plus facile et plus sûr de créer et d'enregistrer une doublure à la place du vrai service.
>
> — https://angular.io/guide/testing-components-scenarios

`TestBed` est une classe de Angular, dont le rôle est de configurer et d'initialiser un environment pour les tests unitaires.

CLI ng et frameworks de test

Par défaut (2020/2021), Angular CLI génère du code de test basé sur *Jasmine* et *Karma*.

- Jasmine, un framework open source pour les tests unitaire en JS : https://jasmine.github.io/
- Karma : *A simple tool that allows you to execute JavaScript code in multiple real browsers*. C'est un outil qui génère un serveur Web qui exécute le code source par rapport au code de test pour chacun des navigateurs connectés. Les résultats de chaque test par rapport à chaque navigateur sont examinés et affichés via la ligne de commande au développeur afin qu'il puisse voir quels navigateurs et tests ont réussi ou échoué. https://github.com/karma-runner/karma/

Le code des tests, appelé également **spécifications**, est placé dans un fichier situé **dans** le dossier du composant et portant le nom du composant postfixé par `.spec.ts`.

On se réfèrera également aux fichiers de configuration à la racine du projet : `angular.json` et `karma.json`

Exemple Hello world !

Afin d'illustrer les instructions de test, nous créons, dans un projet servant d'exemple et via CLI ng, un composant que nous nommerons `hello-word`.

Listing 1. génération d'un composant à l'aide de cli ng

```
$ ng generate component HelloWorld
CREATE src/app/hello-world/hello-world.component.scss (0 bytes)
CREATE src/app/hello-world/hello-world.component.html (26 bytes)
CREATE src/app/hello-world/hello-world.component.spec.ts (655 bytes)
CREATE src/app/hello-world/hello-world.component.ts (295 bytes)
UPDATE src/app/app.module.ts (690 bytes)
$
```

Modifions le template :

```
<h2 class="title">Hello world !</h2>
<p>Nice day {{name}}.</p>
<p>Today it is {{temperature.temp}}°</p>
```

Et le code lié :

Listing 2. src/app/hello-world/hello-world.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  // une interface est un modèle de structure d'objet.
4  // Tout objet (ou classe) se référent à cette interface devra implémenter
5  // ce modèle.
6  interface Temperature {
7    readonly temp : number
8  }
9
10 @Component({
11   selector: 'app-hello-world',
12   templateUrl: './hello-world.component.html',
13   styleUrls: ['./hello-world.component.scss']
14 })
15 export class HelloWorldComponent implements OnInit {
16
17   name : string = "unknown";
18   temperature : Temperature = {temp : 42};
19
20   constructor() { }
21
22   ngOnInit(): void {
23   }
24 }
```

Le composant principal, parent de notre composant, `app.component.html` (avec des classes CSS de bulma) :

Listing 3. app.component.html

```
<style>
.centre {
  text-align: center;
}
</style>
<section class="section centre">
  <div class="container">
    <div class="column ">
      <app-hello-world></app-hello-world>
    </div>
  </div>
</section>
```

à l'exécution nous obtenons :

<component>.spec.ts initial

Via la commande `ng generate component HelloWorld` un script de test est généré : `hello-world.component.spec.ts`.

Dans un premier nous analysons le code généré, puis l'étendrons par ajout de nouveaux tests.

Listing 4. hello-world.component.spec.ts

```
1  import { ComponentFixture, TestBed } from '@angular/core/testing'; ①
2
3  import { HelloWorldComponent } from './hello-world.component';
4
5  describe('HelloWorldComponent', () => {     ②
6    let component: HelloWorldComponent;
7    let fixture: ComponentFixture<HelloWorldComponent>;
8
9    beforeEach(async () => {                  ③
10     await TestBed.configureTestingModule({  ④
11       declarations: [ HelloWorldComponent ]
12     })
13     .compileComponents();
14   });
15
16   beforeEach(() => {                        ⑤
17     fixture = TestBed.createComponent(HelloWorldComponent);
18     component = fixture.componentInstance;
19     fixture.detectChanges();
20   });
21
22   it('should create', function() {          ⑥
23     expect(component).toBeTruthy();         ⑦
24   });
25 });
```

① Import de librairies Angular dédiées aux tests

② Donne un nom à la portée des différents tests *it('xxx')* définis dans ce block, connu sous le nom de **suite** (ou *test suite*)

③ `beforeEach` : Pour définir les actions à exécuter **avant** chacun des tests dans cette suite.

④ Déclaration et compilation du composant (CUT - *Component Under Test*). L'appel de la méthode static `configureTestingModule` de `TestBed` est asynchrone (**promise**), nous demandons d'attendre son retour (`await`) avant créer le composant à l'étape suivante.

⑤ Initialisation du contexte (instanciation du composant et son contexte)

⑥ Vérifie que la variable locale *component* a bien été initialisée

⑦ `toBeTruthy` vérifie que `component` est "not undefined". Une instruction de test de **Jasmine**.

Lancement des tests

La commande CLI pour le lancement des tests est : `ng test`

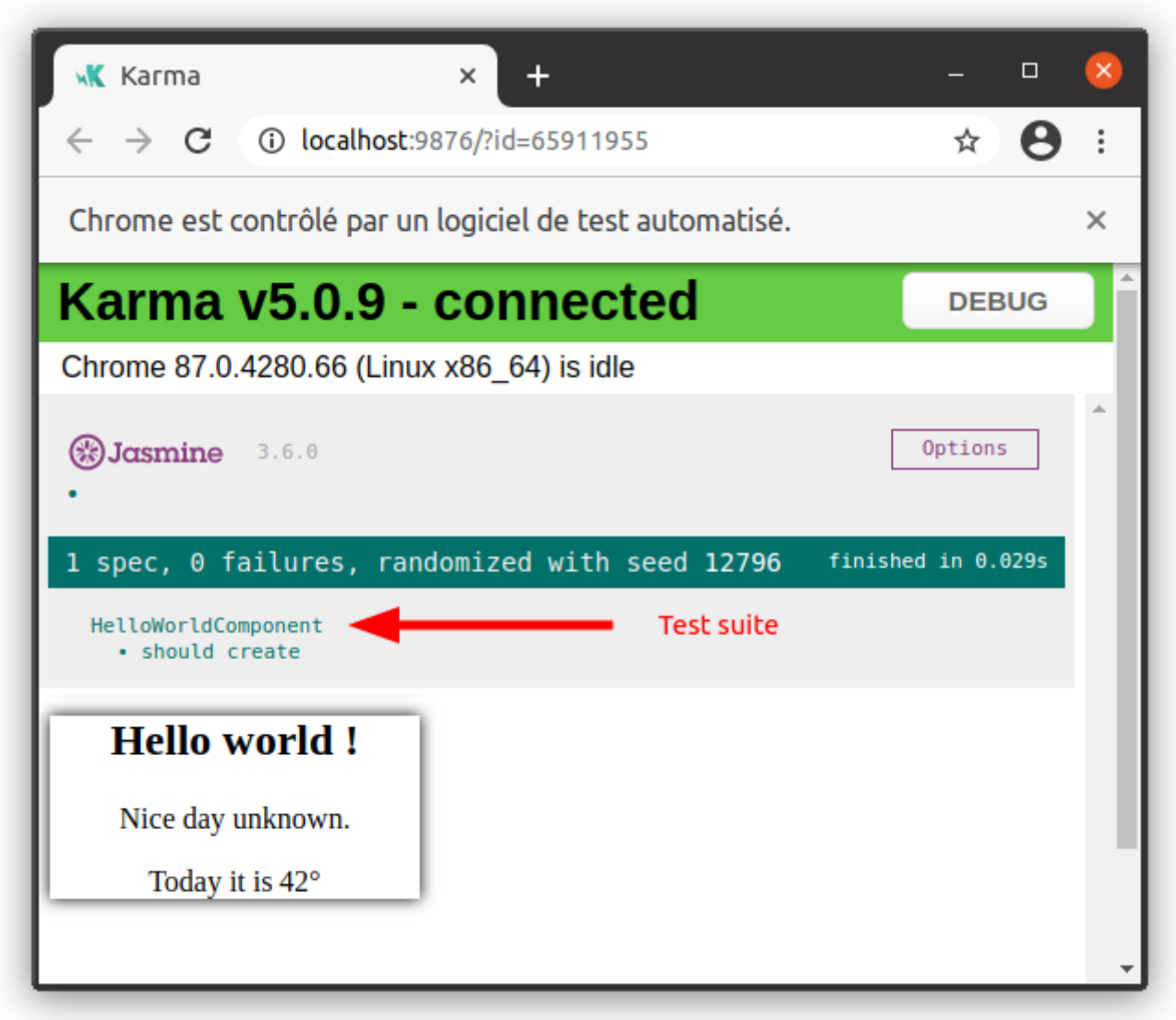
Pour lancer une suite ciblée de tests, la syntaxe consiste à utiliser l'option `include` au lancement :

```
ng test --include=**/someFolder/*.spec.ts
```

*Listing 5. ng test --include=**/hello-world/*.spec.ts*

```
Karma v5.0.9 server started at http://0.0.0.0:9876/
:INFO [launcher]: Starting browser Chrome
:WARN [karma]: No captured browser, open http://localhost:9876/
:INFO [Chrome 87.0.4280.66 (Linux x86_64)]: Connected on socket
Executed 1 of 1 SUCCESS (0.13 secs / 0.057 secs)
TOTAL: 1 SUCCESS
TOTAL: 1 SUCCESS
```

Voici le rapport de test, passé au vert, rendu par l'instance du navigateur support :



Sans surprise, le seul test inclus dans la suite "*HelloWorldComponent*", à savoir *chould create*, est passé !

Il est temps de prendre une pause. Voici un peu de lecture qui vous permettra de découvrir quelques méthodes de tests `toBeGreaterThan`, `toEqual`... : https://jasmine.github.io/tutorials/your_first_suite

Ajout de tests spécifiques

Nous allons maintenant vérifier si le composant se comporte comme attendu, c'est-à-dire s'il est fidèle à ses spécifications (d'où le sufixe `.spec.ts` du fichier regroupant les tests).

Dans un premier temps nous testerons la vue et sa mise à jour avec le modèle (propriété de la classe TS du composant), puis nous ferons évoluer le composant afin qu'il s'appuie sur un service donnant une température.

Test des valeurs par défaut

Vérification de l'état de l'instance gérant les données du modèle

Listing 6. ajout à la "test suite" describe('HelloWorldComponent', ...

```
[...]

it('default name', () => {                ①
  expect(component.name).toBe("unknown"); ②
});
```

① *"defaut name"* est le nom de la spec

② Vérifie la valeur de la propriété *name* de l'objet référencé par *component*

Nous ferons de meme avec la température

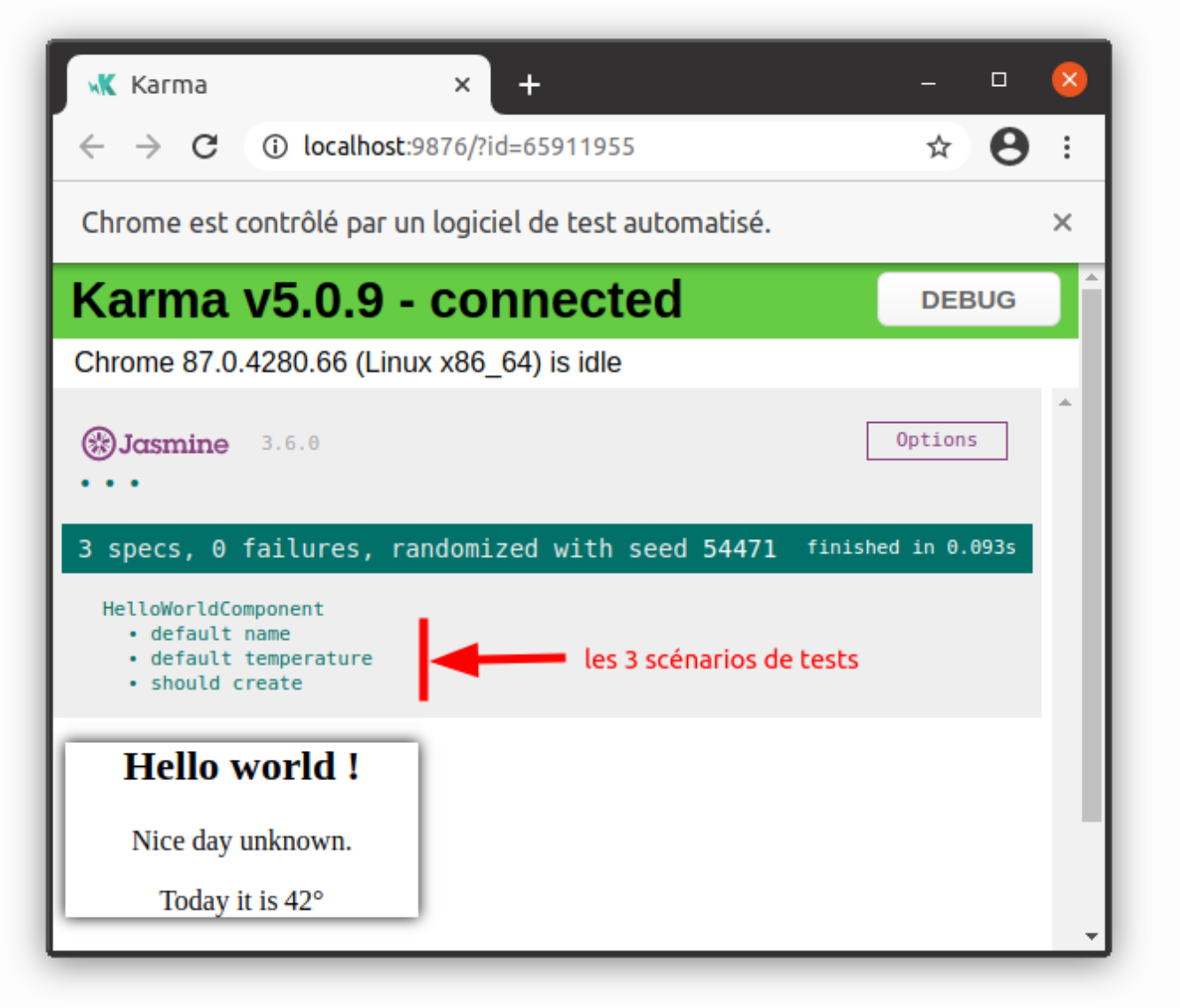
Listing 7. vérifier la temperature par défaut

```
[...]

it('default temperature', () => {
  expect(component.temperature.temp).toBe(42); ①
});
```

① La temperature étant représentée par un objet, nous accédons ici directement à sa propriété public *temp* (qui est en lecture seule)

Vérification (*karma* est normalement toujours actif, et relance les tests après chaque nouvelle compilation)



Nous pouvons vérifier si le composant réagit bien aux changements de valeur de certains de ses propriétés :

Listing 8. vérifier la possibilité de changer 'name'

```
[...]
it('new component.name', () => {
  component.name = "newName";
  expect(component.name).toBe("newName");
});
```

Test de réactivité avec la vue

Nous allons vérifier le lien de réactivité entre le modèle et la vue. Pour cela nous interrogeons la partie de DOM occupée par le composant.

Listing 9. it - view default view name

```
1 it('view default view name', () => {
2   const rootElt = fixture.nativeElement;                                  ①
3   // console.log(rootElt);
4   const firstP = rootElt.querySelector("div p:first-of-type").textContent;  ②
5   expect(firstP).toContain('unknown');                                    ③
6 });
```

① Obtenir le noeud racine du template du composant

② C'est ici que l'on fera usage de **selecteur CSS** pour atteindre les parties souhaitées. Dans le cas présent, on cherche à atteindre le premier `<p>` dans le seul `<div>` du rendu HTML du composant.

③ Vérifie que la chaîne "*unknown*" est inclus dans les texte référencé par `firstP`.

Une bonne connaissance des possibilités des sélecteurs `CSS` s'impose ici. C'est une bonne raison de revisiter des ressources web sur ce sujet, et de s'améliorer ! voir par exemple developer.mozilla Apprendre CSS/Selector et first-of-type selector

Voici un autre test qui vérifie la réactivité de la vue face à un changement de son modèle (instance de la classe du composant).

Listing 10. it - view update name

```
1 it('view update name', () => {
2   const rootElt = fixture.nativeElement;
3   component.name = "newName";   ①
4   fixture.detectChanges();      ②
5   const firstP = rootElt.querySelector("div p:first-of-type").textContent;
6   expect(firstP).toContain('newName');
7 });
```

① Modification dans le modèle

② Demande au contexte de vue du test de se mettre à jour

Exercice

- Concevoir une nouvelle spécification (un nouveau test unitaire) qui vérifie que la température présentée par le composant est bien celle attendue.

Evolution du composant

Recherche d'un service API de requête de température. Il en existe de nombreux, nécessitant la plupart du temps une clé d'accès, et un abonnement au service.

Pour les besoins de ce support, nous utiliserons le service proposé par prevision-meteo.ch.

Testez par vous-même : infos ville de Melun 77000 France

Paramètre de l'application

Nous allons définir un service sous forme d'une classe qui aura la charge de nous fournir une donnée de température.

Pour cela nous commençons par ajouter une classe qui contiendra des données globales, comme l'url de l'API météo.

```
$ ngenerate class GlobalConstants
```

Listing 11. global-constant.ts

```
export class GlobalConstants {
  static readonly meteoUrlAPI : string  = "https://www.prevision-
meteo.ch/services/json/";
}
```

Il est logique que le service météo nous retourne une instance de `Temperature`. Ce type mérite donc d'être déclaré dans un fichier à part. Nous le placerons dans un dossier nommée `model` :

```
ng generate interface Temperature  --path=src/app/model
```

Listing 12. temperature.ts

```
export interface Temperature {
  readonly temp : number
}
```

Parallèlement, nous supprimons la définition de `Temperature` dans `hello-world.component.ts`, et déclarons à la place une dépendance, plus quelques aménagements **temporaires** expliqués ci-après. Le but de ces aménagements temporaires est de faire passer les tests unitaires actuels :

Listing 13. hello-world.component.ts

```
1 import { Component, Input, OnInit } from '@angular/core';
2 import { Temperature } from '../model/temperature';
3 import { MeteoService } from '../service/meteo/meteo.service';
4
5 @Component({
6   selector: 'app-hello-world',
7   templateUrl: './hello-world.component.html',
8   styleUrls: ['./hello-world.component.scss']
9 })
10 export class HelloWorldComponent implements OnInit {
11
12   @Input() name: string = "unknown";
13
14   public get temperature(): Temperature {                    ①
15     return this.meteoService.getTemperatureFromCity("Melun"); ②
16   }
17
18   constructor(private meteoService: MeteoService) { }        ③
19
20   ngOnInit(): void {}
21
22 }
```

① un *getter* est un accesseur automatique (c'est du js). Chaque accès en lecture à *temperature* (sans parenthèses en fin) passera par le code de ce *get*.

② ici, nous appelons le service de *meteoService*.

③ déclaration d'un service qui sera injecté automatiquement par Angular, en tant qu'attribut privé (*property*) de la classe.

Début de creation du service météo (première version)

Nous placerons ce service dans un dossier dédié (via `cli ng`) :

```
ng generate service Meteo --flat=false --path=src/app/service
```

Listing 14. src/app/service/meteo/meteo.service.ts

```
import { Injectable } from '@angular/core';
import { Temperature } from 'src/app/model/temperature';

@Injectable({
 providedIn: 'root'
})
export class MeteoService {

  getTemperatureFromCity(city : string) : Temperature {
    return {temp : 42};
  }

  constructor() { }
}
```

À ce niveau d'avancement, les précédents tests unitaires devraient tous passer.

Nous venons de réaliser un ***refactoring***. C'est très souvent le pris à payer pour favoriser les évolutions à venir.

Update de l'interface Temperature

Dans l'interface `Temperature` nous ajoutons le nom de la ville concernée.

Listing 15. src/app/model/temperature.ts

```
export interface Temperature {
  readonly temp : number
  readonly city : string
}
```

et bien entendu, le composant présente cette valeur à l'utilisateur :

Listing 16. src/app/hello-world/hello-world.component.html

```
<style>
.box {
  box-shadow: inset 0 0 1em white, 0 0 .5em black;
  text-align: center;
  width: 200px;
}
</style>

<div class="box">
  <h2 class="title">Hello world !</h2>
  <p>Nice day {{name}}.</p>
  <p>Today at {{temperature.city}} it is {{temperature.temp}}°</p>
</div>
```

Fort heureusement TpyeScript étant typé, les erreurs révélées lors de la compilation vous guide vers les parties à mettre à jour suite à la modification de l'interface.

Par exemple, nous ne manquerons pas de mettre à jour la structure de l'objet retourné par la méthode du service méto :

Listing 17. src/app/service/meteo/meteo.service.ts

```
[...]

 getTemperatureFromCity(city : string) : Temperature {
    return {city: "Melun", temp : 42};
 }
```

À ce moment des modifications, la suite de tests `HelloWorldComponent` devrait passer.

Nous allons maintenant demander au service d'effectuer une requête http à la demande d'une température pour une ville donnée.

Listing 18. src/app/service/meteo/meteo.service.ts

```
1 import { Injectable } from '@angular/core';
2 import { GlobalConstants } from 'src/app/global-constants'
3 import { Observable, of } from 'rxjs';
4 import { HttpClient } from '@angular/common/http';
5
6 @Injectable({
7   providedIn: 'root'
8 })
9 export class MeteoService {
10
11   constructor(private httpClient: HttpClient) { }          ①
12
13   getTemperatureFromCity(city: string): Observable<any> {  ②
14     return this.httpClient.get(this.getBaseUrl() + "/" + city);
15   }
16
17   getBaseUrl() : string {                                  ③
18     return GlobalConstants.meteoUrlAPI;
19   }
20 }
```

① Définition d'une dépendance à une instance de `HttpClient` afin de pouvoir lancer des requêtes HTTP depuis la logique de l'application.

② Lancement de la requête HTTP, qui retourne un objet de type `Observable` *non typé* ; si vous êtes sûr de la structure de la réponse, vous pouvez typer le résultat plus finement en mentionnant à la place de *any* le nom d'une classe ou d'une interface.

③ Afin de connaitre, dans les tests, l'url utilisée.

Nous voici avec un service qui dépend d'un autre service avec appel asynchrone... Pas simple à tester.

Heureusement, Angular vient avec des bibliothèques de classes dédiées à ce problème, qui nous permettra de simuler des requêtes HTTP.

Listing 19. src/app/service/meteo/meteo.service.spec.ts

```
1 import { HttpTestingController, HttpClientTestingModule } from
'@angular/common/http/testing';
2 import { TestBed } from '@angular/core/testing';
3 import { MeteoService } from './meteo.service';
4
5 // voir : https://angular.io/guide/http#testing-http-requests
6
7 describe('MeteoService', () => {
8   let service: MeteoService;
9   let mockHttpTestingController: HttpTestingController;  ①
10
11   beforeEach(() => {
```

```
12    TestBed.configureTestingModule({
13      imports: [HttpClientTestingModule],
14      providers: [MeteoService]
15    });
16    service = TestBed.inject(MeteoService);
17    mockHttpTestingController = TestBed.inject(HttpTestingController); ②
18  });
19
20  it('should be created', () => {
21    expect(service).toBeTruthy();                          ③
22  });
23
24  it('city known - Melun with temperature', () => {
25    let testUrl = service.getBaseUrl() + "/melun";         ④
26
27    service.getTemperatureFromCity("melun").subscribe(data => {
28                                                                    ⑤
29      console.log("TEST RECEIVE TEMP : " + data?.current_condition?.temp);
30      expect(data?.current_condition?.temp).toBeGreaterThan(10);
31      expect(data?.city_info?.name).toEqual("Melun");
32      expect(data?.city_info?.country).toEqual("France");
33    });
34
35    const req = mockHttpTestingController.expectOne(testUrl);
36    expect(req.request.method).toEqual('GET');
37
38    let mockedResponse = {   ⑥
39        city_info: {
40          name: "Melun",
41          country: "France"
42        },
43        current_condition: {
44          temp: 13
45        }
46    };
47
48    req.flush(mockedResponse); ⑦
49
50    mockHttpTestingController.verify();  ⑧
51  });
52
53 });
```

① Déclaration d'une référence à une instance de mock http

② Demande d'injection et récupération de l'instance

③ Le service existe t'il ?

④ Le url utilisées par le commosant et le mock http doivent être identiques

⑤ Lorsque que la requête reçoit une réponse, les *souscripteurs* (les observateurs) en sont avisés. La donnée reçue est donnée comme argument (nommée **data**) de la fonction callback anonyme

passée à la méthode `subscribe` : on peut donc ici coder les tests avec `expect`. On remarquera l'usage `?.` qui est une façon sûre (*safe*) de tenter d'atteindre un symbol dans une structure que nous ne maitrisons pas (rend `undefined` en cas d'échec). (voir https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html)

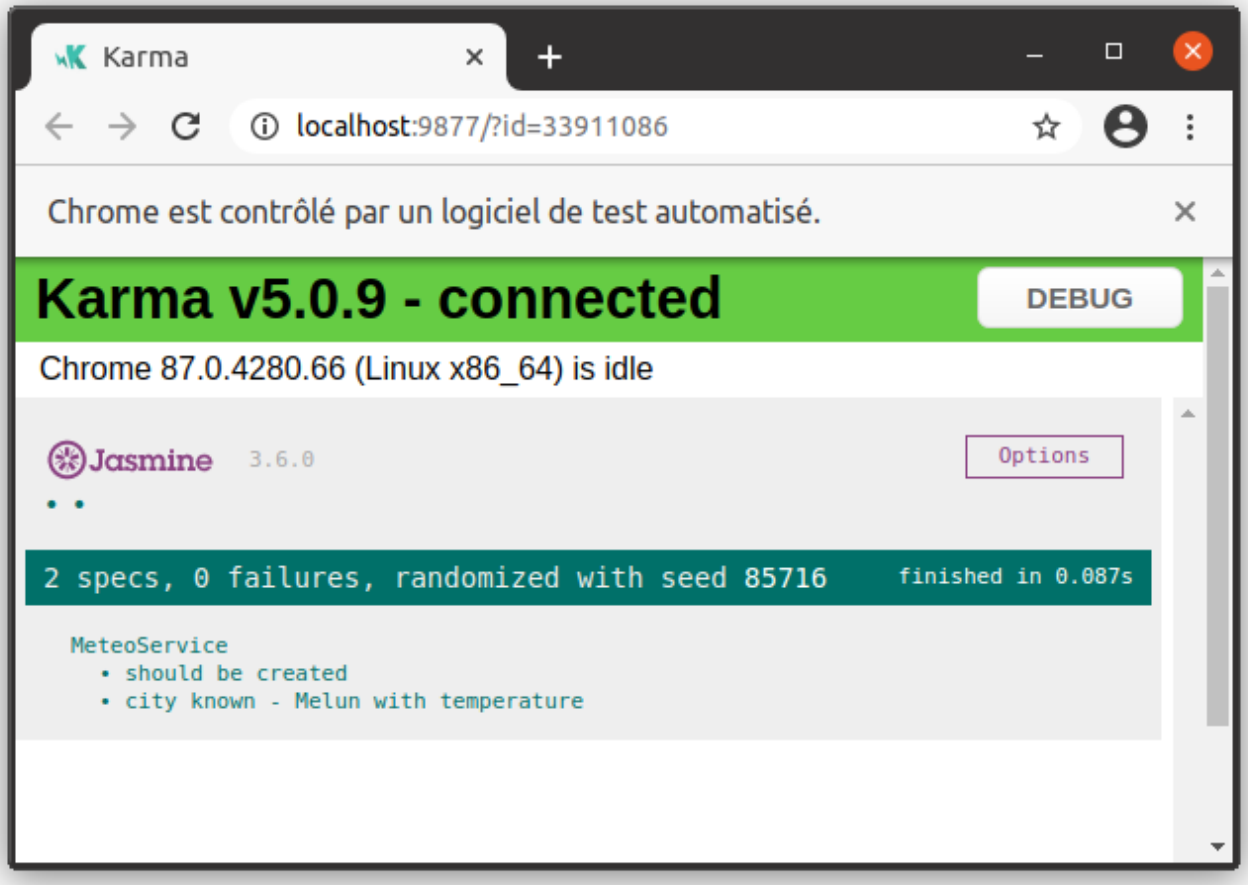
⑥ Construction des données simulant la réponse

⑦ Lance la réponse, **ce qui déclenchera la méthode resolve de l'observable** (*ses abonnés en seront avisés*)

⑧ Vérifie qu'il n'y a pas de demandes en suspens.

Vérification

```
ng test --include=**/service/meteo/*.spec.ts
```



Conclusion

Nous avons présenté comment mettre en oeuvre des tests unitaires avec Angular, et par là même, via une méthodologie basée sur le refactoring.

Exercices

- Modifier l'interface `Temperature` afin qu'elle détienne le nom du pays (voir le json reçu du service API). Bien entendu des modifications devront être apportées ici et là dans le code. Faire passer les tests.
- Modifier de nouveau l'interface `Temperature` afin de présenter la température **maximale** et la **minimale** du jour.
- Défi : Donner la tendance de la météo (une analyse basée sur les données des jours à venir, données reçues avec la réponse)