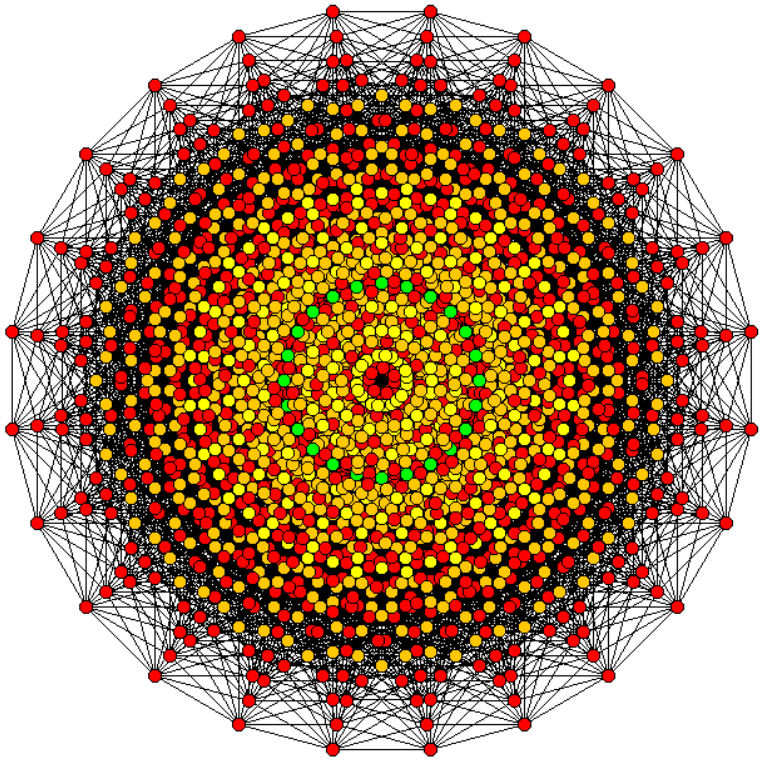


Test unitaire

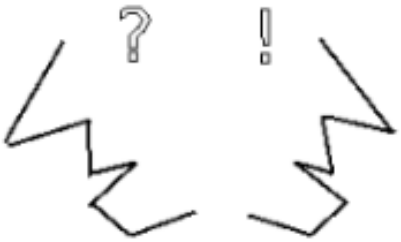
O. Capuozzo

Version 1.1, 2021-11-18

Table des matieres

Présentation . . . 1
Prise en main de la gestion des tests avec Angular . . . 1
- Prérequis . . . 1
- Contexe Angular . . . 1
- Tester un composant . . . 2
- CLI ng et frameworks de test . . . 3
- Création de l'application . . . 3
- Composant Hello world ! . . . 4
- <component>.spec.ts . . . 6
- Lancement des tests . . . 7
- Ajout de tests spécifiques . . . 9
  - Test des valeurs par défaut . . . 9
  - Test de réactivité avec la vue . . . 11
  - Exercice . . . 12
- Evolution du composant . . . 12
  - Paramètre de l'application . . . 12
- Création d'un service de météo . . . 14
- Update de l'interface Temperature . . . 17
  - Exercice . . . 18
  - Mise à jour du test du service meteo . . . 18
- Appel d'un service API via HTTP . . . 19
- Vérification . . . 22
- Evolution de l'application . . . 22
- Conclusion . . . 26
  - Exercices . . . 26

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

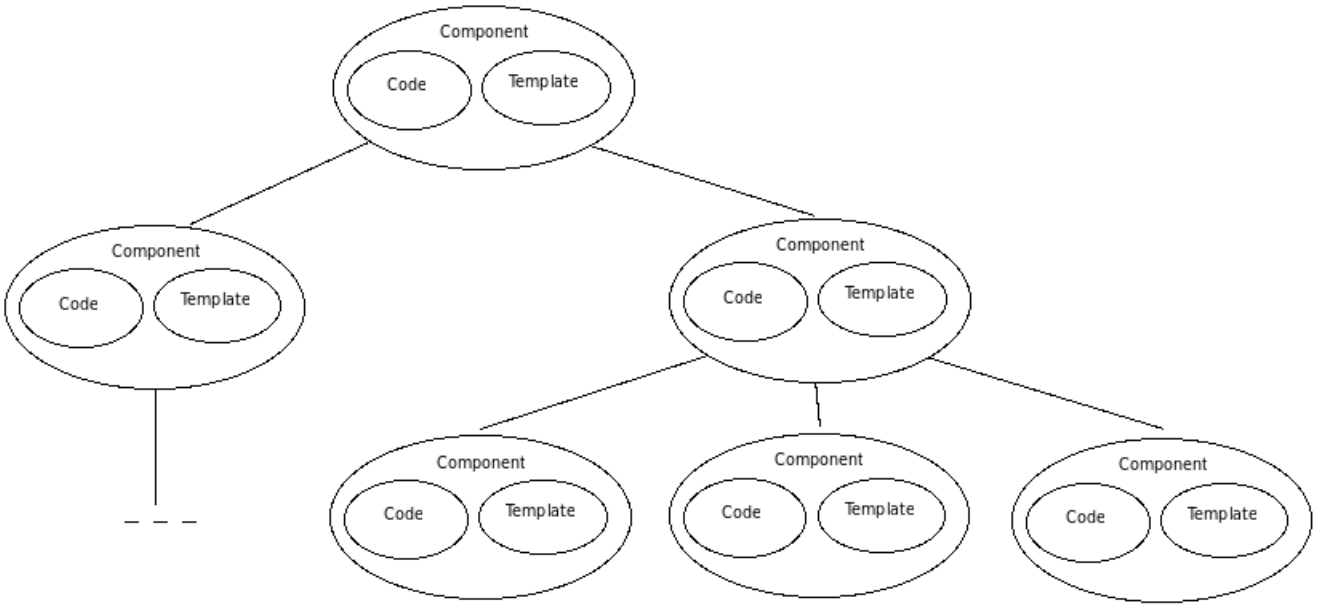
Contexe Angular

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 des services 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 (2021/2022), 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`

Création de l'application

Listing 1. génération du projet basique à l'aide de cli ng

```
$ ng new projet-demo
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? CSS
[...]
   Packages installed successfully.
    Successfully initialized git
$
```

On installe un framework `css` :

```
npm install bulma --save
```

puis mettons a jour `angular.json`:

```
 "styles": [
   "node_modules/bulma/css/bulma.min.css",
   "src/styles.css"
 ],
```

Composant Hello world !

Afin d'illustrer les instructions de test, nous créons, dans le projet demo, un composant que nous nommerons `hello-world` qui, en plus de ire bonjour nous donnera la température du jour.

```
.génération d'un composant à l'aide de cli ng
$ ng generate component HelloWorld
CREATE src/app/hello-world/hello-world.component.css (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 (294 bytes)
UPDATE src/app/app.module.ts (414 bytes)
$
```

Modifions le template du composant :

```
<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 it is {{temperature.current_condition.tmp}}°</p>
</div>
```

Et le code lié :

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

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

/// une interface est un modèle de structure d'objet (un type personnalisé)
// Tout objet (ou classe) se référent à cette interface
// devra, au moins, implémenter ce modèle (basé sur une API vue plus loin)
 interface Temperature {
    current_condition: {
     tmp: number
    }
 }

@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
export class HelloWorldComponent implements OnInit {

  // attributs de la classe
  name : string = "unknown";

  temperature: Temperature = {
    current_condition: {
      tmp: 42
    }
  };

  constructor() { }

  ngOnInit(): void {
  }
}
```

Le template `app.component.html`, du composant principal, utilise notre composant, (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> ①
    </div>
  </div>
</section>
```

<1>: Le composant `HelloWorldComponent` est un composant enfant de `AppComponent`.

à l'exécution nous obtenons :

<component>.spec.ts

Lorsque nous avons demandé la création du composant `HelloWorld` par 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 (`HelloWorldComponent`) 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, ce qui assure l'indépendance entre les tests.

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. Les instructions du test sont placées dans le corps d'une fonction anonyme.

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`

Cette commande fera échouer le test nommé : `should render title'` de `app.component.spec.ts`. Ceci est tout à fait normal car nous avons supprimé le comportement par défaut du template du

composant principal.

Pour résoudre ce problème, mettre en commentaire le test unitiare en question :

Listing 5. app.component.spec.ts

```
 1 [...]
 2
 3 /*
 4 it('should render title', () => {
 5   const fixture = TestBed.createComponent(AppComponent);
 6   fixture.detectChanges();
 7   const compiled = fixture.nativeElement as HTMLElement;
 8   expect(compiled.querySelector('.content span')?.textContent).toContain('projet-
demo app is running!');
 9 });
10 */
```

Pour lancer une suite ciblée de tests, la syntaxe consiste à utiliser l'option `include` au lancement : `ng test --include=**/someFolder/*.spec.ts`

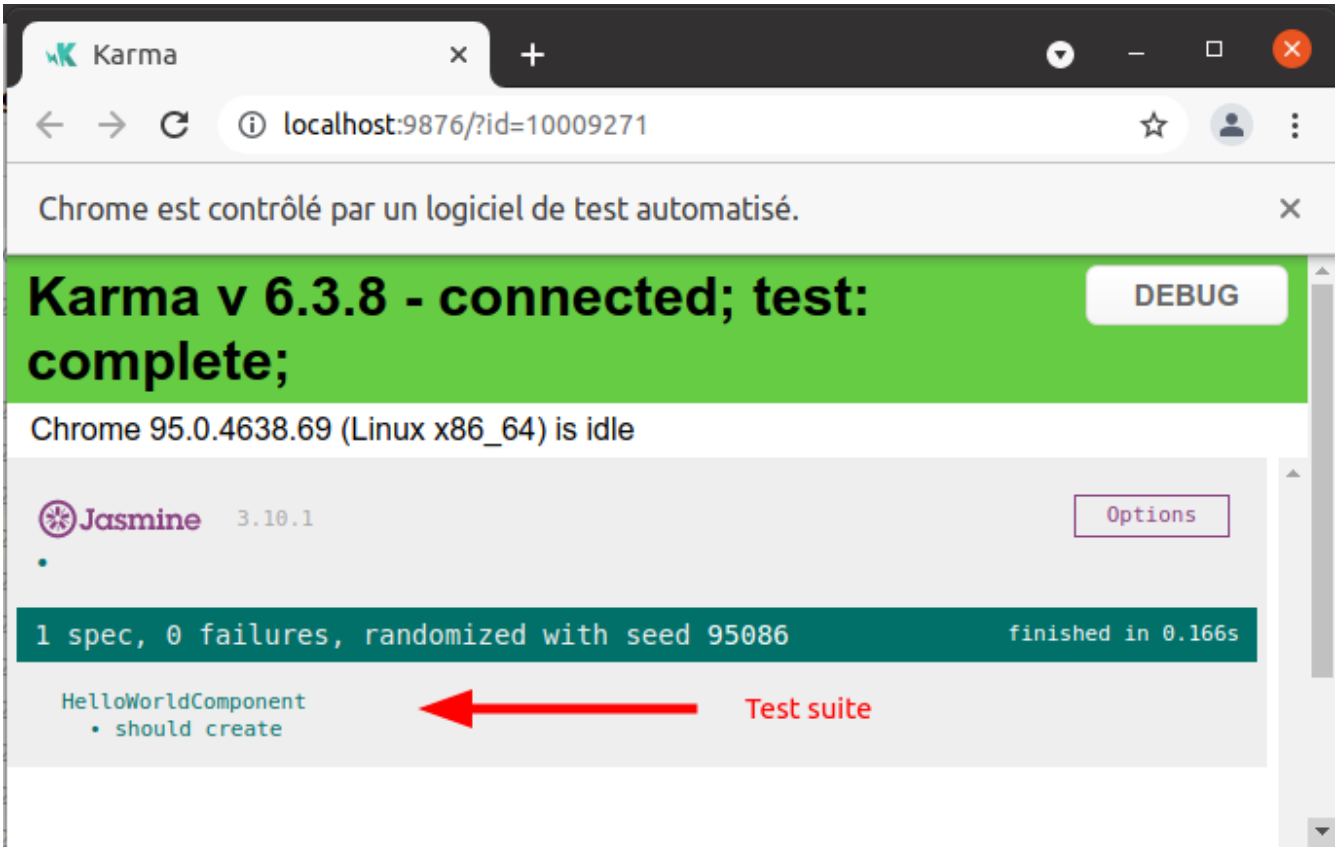
Nous lancerons donc le seul test actuel :

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

*Listing 6. 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 7. ajout à la "test suite" describe('HelloWorldComponent', ...

```
[...]

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

① *"defaut name"* est le nom du test, comme une variable ou une méthode, son nom doit clairement communiquer son objectif.

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

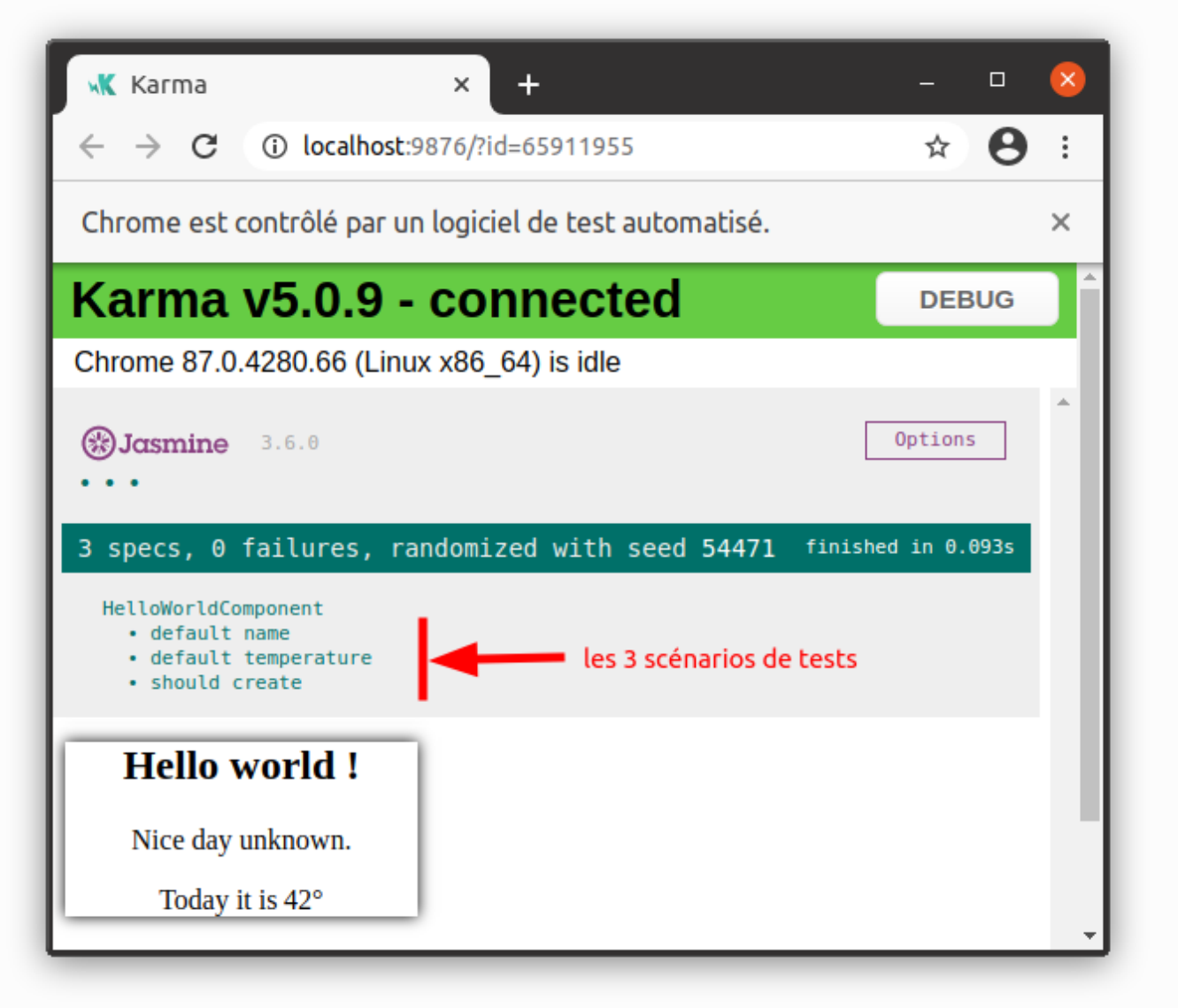
Nous ferons de meme avec la température

Listing 8. vérifier la temperature par défaut

```
[...]

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

① La temperature étant représentée par un objet, nous accédons ici directement à sa propriété public *current_condition.tmp*

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



Nous allons maintenant vérifier que le composant réagit bien aux changements de valeur de

certaines de ses propriétés :

Listing 9. 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 10. 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

Nous vous invitons à tester en placant un identifiant d'élément du DOM dans le template du composant, comme par exemple :

```
<p id="hello">Nice day {{name}}.</p>
```

Dans ce cas, le texte de l'élément est atteint sans ambiguité ainsi :

```
rootElt.querySelector("#hello").textContent;
```

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 11. 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 automatisé) 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 avec un navigateur, et consulter le **format JSON** de la réponse : 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.

```
$ ng generate class GlobalConstants
```

Listing 12. global-constant.ts

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

Il serait 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 nommé `model` :

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

Listing 13. temperature.ts

```
export interface Temperature {
    current_condition: {
     tmp: number
    }
}
```

Définir une interface c'est renforcer la cohérence de type dans votre programme et permettre à TypeScript de détecter des erreurs à la conception (lors de la compilation, donc bien avant l'exécution).

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 14. hello-world.component.ts

```
import { Component, Input, OnInit } from '@angular/core';
import { Temperature } from '../model/temperature';

@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.scss']
})
export class HelloWorldComponent implements OnInit {
  // attributs de la classe
  @Input() name: string = "unknown";

  temperature: Temperature = {
    current_condition: {
      tmp: 42
    }
  };

  constructor() { }

  ngOnInit(): void {}

}
```

À ce niveau d'avancement, les précédents tests unitaires devraient tous passer. Vérifiez-le, et n'avancez pas plus loin tant que votre projet support n'est pas stabilisé.

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

Création d'un service de météo

Dans un premier, arrêtez l'exécution automatique des tests de `hello-world.component`, car les modifications que nous allons apporter à notre composant casseront la logique de ces tests, que nous corrigerons ensuite. (`CTRL+c dans le terminal où a été lancé la commande`)

Nous allons créer un service qui va interroger l'API météo. Nous découpons ce refactoring en deux parties :

1. Conception de la classe du service, puis test d'utilisation
2. Appel du service distant via son API avec retour JSON

Nous placerons ce service dans une classe dédiée (via `cli ng`) :

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

Listing 15. src/app/service/meteo/meteo.service.ts

```
1 import { Injectable } from '@angular/core';
2 import { Observable, of } from 'rxjs';
3 import { Temperature } from 'src/app/model/temperature';
4
5 @Injectable({
6  providedIn: 'root'  ①
7 })
8
9 export class MeteoService {
10
11   // retourne toujours le même objet observable, pour commencer
12   getTemperatureFromCity(city : string) : Observable<Temperature>  { ②
13     return of(
14       {
15        current_condition: {
16          tmp: 42
17        }
18       });
19   }
20
21   constructor() { }
22 }
```

① Cet attribut du décorateur (annotation) `@Injectable` spécifie que le composant est injectable sur l'ensemble de l'application

② Remarquez que l'objet `Observable` retourné est bien compatible avec l'interface `Temperature` (le

type de la méthode)

Nous passons par un objet `Observable` car l'appel de l'API (à venir) est suceptible d'échouer.

Puis nous déclarons l'utiliser dans le composant `HelloWorldComponent`.

Listing 16. 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'; ①
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   temperature: Temperature = {
15     current_condition: {
16       tmp: 42
17     }
18   };
19
20    myObserver = {
21     next: (data: Temperature) => {
22       if (data?.current_condition?.tmp) {
23         this.temperature.current_condition.tmp = data.current_condition.tmp;
24       } else {
25         this.temperature.current_condition.tmp = -42;
26       }
27     }
28     ,
29     error: (err: Error) => console.error('Observer got an error: ' + err),
30     complete: () => console.log('Observer got a complete notification'),
31   }
32
33   public getActualTemperature() {
34     // La méthode getTemperatureFromCity retourne un Observable,
35     // Nous nous abonnons en tant qu'observateur, via un objet dédié.
36     // Voir https://angular.io/guide/observables
37     let cityName : string = "melun";
38     this.meteoService.getTemperatureFromCity(cityName).subscribe(this.myObserver
);②
39   }
40
41   constructor(private meteoService: MeteoService) { }         ③
42
43   ngOnInit(): void {}
44
45 }
```

① déclaration de la dépendance à `MeteoService`

② ici, nous appelons le service de *meteoService*. Actuellement ce service n'exploite pas son paramètre, mais il en attend un quand même.

③ déclaration d'un service qui sera injecté automatiquement par Angular, et automatiquement déclaré en tant qu'attribut privé (*property*) de la classe (2 en 1).

À ce niveau d'avancement, les précédents tests unitaires devraient tous passer. Vérifiez-le, et n'avancez pas plus loin tant que votre projet support n'est pas stabilisé.

Update de l'interface Temperature

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

Listing 17. src/app/model/temperature.ts

```
export interface Temperature {
    city_info: {
        name: string
    },
    current_condition: {
        tmp: number
    }
}
```

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

Listing 18. 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 id="tempcity">Today it is {{temperature.current_condition.tmp}}° in
{{temperature.city_info.name}}</p>

</div>
```

Fort heureusement TypeScript 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 19. src/app/service/meteo/meteo.service.ts

```
[...]

getTemperatureFromCity(city : string) : Observable<Temperature>  { ②
    return of(
      {
       city_info: {
        name: "Melun",
       },
       current_condition: {
         tmp: 42
       }
      });
  }
```

Exercice

- 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.

Mise à jour du test du service meteo

Étant donné que l'interface du service à changer, nous devons maintenant modifier la logique de nos test unitaires, et déclarer une dépendance vers le service de meteo.

Listing 20. meteo.service.spec.ts

```
import { TestBed } from '@angular/core/testing';
import { MeteoService } from './meteo.service';  ①

describe('MeteoService', () => {
 let service: MeteoService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [MeteoService]
    });
    service = TestBed.inject(MeteoService);  ②
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('city Melun - with 42 temperature', () => {
    let testUrl = service.getBaseUrl() + "/paris";

    service.getTemperatureFromCity("melun").subscribe(data => { ③
      console.log("TEST RECEIVE TEMP : " + data?.current_condition?.tmp);
      expect(data?.current_condition?.tmp).toEqual(42);
      expect(data?.city_info?.name).toEqual("Melun"); ④

    });

  });
```

① Déclaration de la dépendance au service meteo

② Demande d'instanciation du service (on fournit ici une classe d'implémentation)

③ Les tests sont réalisés au moment de la réception normale de la réponse (c'est le minimum ici)

④ Dans l'hypothèse où vous avez réalisé l'exercice demandé ci-dessus.

Appel d'un service API via HTTP

Nous allons maintenant demander au service de nous retourner une réponse à partir d'un appel à un service HTTP dont l'application devra dépendre. La réponse via un `Observable` s'explique par le fait que le retour du service dépend de facteurs externes : qualité de la bande passante, du réseau, disponibilité du service demandé, etc.

Listing 21. src/app/service/meteo/meteo.service.ts

```
import { Injectable } from '@angular/core';
import { GlobalConstants } from 'src/app/global-constants'
import { Observable, of } from 'rxjs';
import { HttpClient } from '@angular/common/http';

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

  constructor(private httpClient: HttpClient) { }          ①

  getTemperatureFromCity(city : string) : Observable<Temperature>  { ②
    return this.httpClient.get<Temperature>(this.getBaseUrl() + "/" + city);
  }

  getBaseUrl() : string {                                  ③
    return GlobalConstants.meteoUrlAPI;
  }
}
```

① Définition d'un attribut (une instance de `HttpClient`) afin de pouvoir lancer des requêtes HTTP depuis ce service.

② Lancement de la requête HTTP, qui retourne un objet de type `Observable` typé.

③ On passe par une méthode pour obtenir l'url appelée, ce qui nous permettra de connaitre, dans les tests, l'url de l'API utilisée.

Nous voici avec un service qui dépend d'un autre service… 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 22. src/app/service/meteo/meteo.service.spec.ts

```
import { HttpTestingController, HttpClientTestingModule } from
'@angular/common/http/testing';
import { TestBed } from '@angular/core/testing';
import { MeteoService } from './meteo.service';

// voir : https://angular.io/guide/http#testing-http-requests

describe('MeteoService', () => {
  let service: MeteoService;
  let mockHttpTestingController: HttpTestingController;  ①

  beforeEach(() => {
    TestBed.configureTestingModule({
      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 Paris - with 10 temperature', () => {
25      let testUrl = service.getBaseUrl() + "/paris"; ④
26
27      let mockedResponse : Temperature = { ⑤
28        city_info: {
29          name: "Paris",
30          country: "France"
31        },
32        current_condition: {
33          tmp: 10
34        }
35      };
36
37      service.getTemperatureFromCity("paris").subscribe(data => {  ⑥
38        console.log("TEST RECEIVE TEMP : " + data?.current_condition?.tmp);
39        expect(data?.current_condition?.tmp).toEqual(10);
40        expect(data?.city_info?.name).toEqual("Paris");
41        expect(data?.city_info?.country).toEqual("France");
42      });
43
44      const req = mockHttpTestingController.expectOne(testUrl); ⑦
45      // expect(req.request.method).toEqual('GET');
46      req.flush(mockedResponse); ⑧
47      // mockHttpTestingController.verify();
48    });
49
50 });
```

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 composant et le mock http doivent être identiques

⑤ Construction des données simulant la réponse

⑥ 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)

⑦ Début de lancement d'une requête HTTP

⑧ Injecte la réponse, **ce qui déclenchera la méthode resolve de l'observable** (*ses abonnés en seront avisés*)

Vérification

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

Evolution de l'application

Maintenant que les tests sont passés, nous pouvons nous focaliser sur l'interface utilisateur, l'UX.

On proposera à l'utilisateur de saisir un nom de ville, et l'application mettra automatiquement à jour la température.

Listing 23. 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: 250px;
    }
</style>

<div class="box">
    <h2 class="title">Hello world !</h2>
    <p id="hello">Nice day {{name}}.</p>
    <p id="tempcity">Today it is {{temperature.current_condition.tmp}}° in
{{temperature.city_info.name}}</p>
    <label for="ville">Ville : </label>
    <input id="ville" [value]="temperature.city_info.name"  ①
                      (input)="setTemperatureCity($event)" size="8" >  ②
</div>
```

① On présente un composant HTML `input` en liant son attribut `value` à un attribut de la variable d'instance de notre composant déclaré dans le `ts` (nom de la ville).

② On lie l'évenement `change` de l'input à un `handler` (une méthode de notre composant). Ainsi, chaque fois que la valeur du input change, cette méthode est appelée, avec `$event` en argument. Ce `$event` désigne un `DOM event object` pointant (`target) vers le composant cible, l'input ici, permettant à la méthode d'exploiter sa value.

Voici le code du composant correspondant.

Listing 24. src/app/hello-world/hello-world.component.html

```
import { Component, Input, OnInit } from '@angular/core';
import { Temperature } from '../model/temperature';
import { MeteoService } from '../service/meteo/meteo.service';


@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
export class HelloWorldComponent implements OnInit {

  // attributs de la classe
  name: string = "unknown";
  temperature: Temperature = {
    city_info: {
      name: "Melun",
      //country: "France"
    },
```

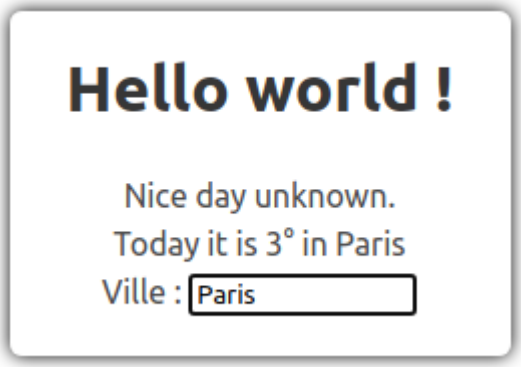
```
    current_condition: {
      tmp: 42
    }
  };

  myObserver = {
    next: (data: Temperature) => {
      if (data?.city_info?.name) {
        console.log("in observer : " + JSON.stringify(data.city_info.name));
      }
      // y a t-il une température dans la réponse du service ?
      if (data?.current_condition?.tmp) {
        this.temperature.current_condition.tmp = data.current_condition.tmp;
      } else {
        this.temperature.current_condition.tmp = -42;
      }
    }
    ,
    error: (err: Error) => console.error('Observer got an error: ' + err),
    complete: () => console.log('Observer got a complete notification'),
  };


  public updateTemperature() {  ①
    console.log(`Ville : ${this.temperature.city_info.name}`);

    // La méthode getTemperatureFromCity retourne un Observable,
    // Nous nous abonnons en tant qu'observateur, via un objet dédié.
    // Voir https://angular.io/guide/observables
    const cityName : string = this.temperature.city_info.name; ②
    this.meteoService.getTemperatureFromCity(cityName).subscribe(this.myObserver);
③

  }

  setTemperatureCity(event: Event) {
    const cityName = (event.target as HTMLInputElement).value;
    if (cityName.length > 3) {
      this.temperature.city_info.name = cityName;
      this.updateTemperature();
    }
  }

  // déclaration d'un service qui sera injecté automatiquement par Angular,
  // en tant qu'attribut privé (_property_) de la classe.
  constructor(private meteoService: MeteoService) { }

  ngOnInit(): void {
    // instruction d'initialisation du composant ici
  }
}
```

① Cette méthode appelle le service météo en vue d'obtenir la température de la ville saisie par l'utilisateur

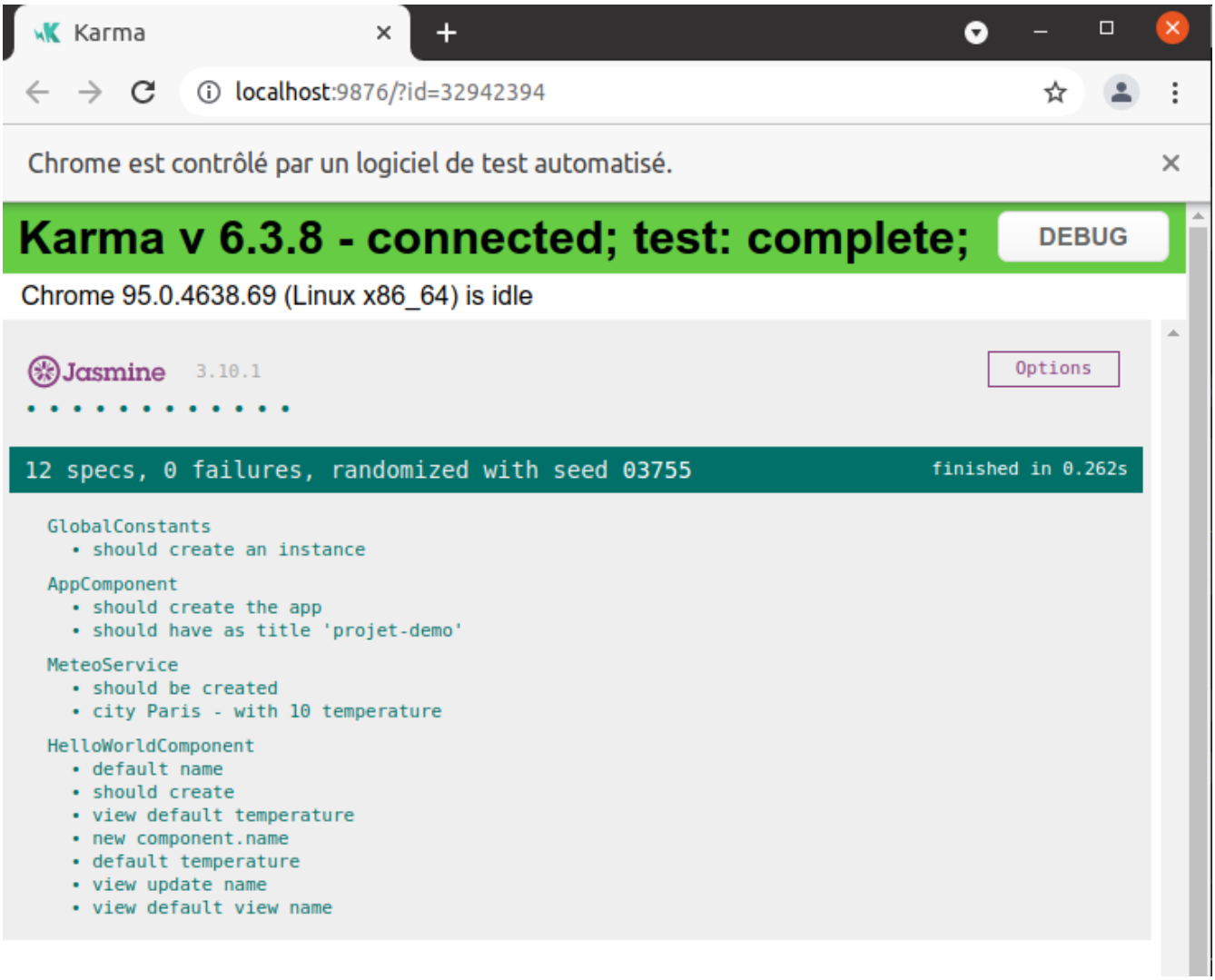
② Place le nom de la ville dans une variable pour une meilleure lecture du code.

③ Demande au service météo (via un obervateur)

Exemple de résultat (`ng serve`) le 27 nov 2021 à 19h30 :



Nos tests unitaires sont toujours opérationnels, sans erreurs et sans echecs.



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

- Si ce n'est pas encore fait, 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)