TP's

Angular Module

Bewizyu, Nicolas Hodicq

Version 1.0, 2018-09-16

Table des matières

Chapitre 1. TypeScript

1.1. Installation

TypeScript est un module global Node.js.

- Installez le module global node typescript

```
npm install -g typescript
```

- Vérifiez que Typescript est bien installé. En ligne de commande TypeScript s'exécute via la ligne de commande `tsc`

```
// Récupérer la version
tsc -v

// Aide
tsc --help
```

1.2. Get started

1.2.1. Premier script TypeScript

Vous pouvez vous aider de cette annexe : typescript et du site officiel

- Créez un dossier de travail et se placer dedans
- Création du fichier de configuration de Typescript, nous allons utiliser la même configuration que celle que vous trouverez dans un projet Angular. Dans un terminal:

```
    // Configuration de base pour un projet Angular
    tsc --init --target es5 --sourceMap --experimentalDecorators
--emitDecoratorMetadata
```

- Un fichier `tsconfig.json` est créé dans le dossier. Ce fichier permet de configurer TypeScript, par exemple, vers quelle version de JavaScript fichier TypeScript doit-il être transpilé.
- Créez un fichier `first-script.ts`
- Dans le terminal, lancez une tâche TypeScript (TS) qui 'transpilera' les fichiers TS vers du JS à chaque modification dans les fichiers TS du dossier.

```
    tsc --watch
```

- Déclarez une variable `age` de type number et assignez lui la valeur 30.
- Assignez ensuite (ligne suivante) une nouvelle valeur: '30'
- Que constatez vous ?
- TypseScript vous garantit que le type d'une variable sera celui que vous avez déclaré.
- Changez le type de age pour que cela soit un number ou un string
- Vous ne devriez plus avoir d'erreur au niveau de la nouvelle assignation en string
- Loguez la variable age

1.2.2. Exécuter le premier script

Si vous regardez dans le dossier du TP précédent, vous pouvez constatez qu'en plus du fichier first-script.ts, vous avez deux fichiers:

- first-script.js : Fichier 'transpilé' en fonction de la configuration TypeScript
- first-script.js.map: Fichier de mapping entre les sources et le js transpilé

Sourcemap [Mozilla documentation]

Les sources JavaScript exécutées par le navigateur sont souvent différentes des sources originales crées par un développeur. Par exemple :

- Les sources sont souvent combinées et minifiées afin d'optimiser le temps que met le serveur à les fournir.
- Le JavaScript d'une page est souvent généré automatiquement. Par exemple lorsqu'il est compilé depuis un langage comme CoffeeScript ou TypeScript.

Dans ces situations, il est bien plus facile de déboguer le code dans son état original plutôt que dans celui utilisé par le navigateur. Une source map est un fichier grâce auquel le débogueur peut faire le lien entre le code étant exécuté et les fichiers sources originaux, permettant ainsi au navigateur de reconstruire la source originale et de l'afficher dans le Débogueur.

Par exemple voici ce que vous obtiendrez pour ce fichier ts:

Sources: 1. first-script.ts

```
let maVariable: string = 'Module formation Angular';

let age:number|string = 40;
age = '40'

console.log(`Age: ${age}`);
```

Sources: 2. first-script.js

```
"use strict";
var maVariable = 'Module formation Angular';
var age = 40;
age = '40';
console.log("Age: " + age);
//# sourceMappingURL=first-script.js.map
```

Sources: 3. first-script.js.map

```
{"version":3,"file":"first-script.js","sourceRoot":"","sources":["first-script.ts"
],"names":[],"mappings":";AAAA,IAAI,UAAU,GAAW,0BAA0B,CAAC;AAEpD,IAAI,GAAG,GAAiB,EAAE,C
AAC;AAC3B,GAAG,GAAG,IAAI,CAAA;AAEV,OAAO,CAAC,GAAG,CAAC,UAAQ,GAAK,CAAC,CAAC"}
```

Node.js

- Node.js ne sait pas comment exécuter un script TypeScript. Sans outil additionnel, vous devez le faire avec le fichier javascript généré.

- Exécutez le script JS

```
node first-script.js
```

ts-node

- ts-node est un module global Node.js permettant d'exécuter directement un fichier TS.
- Installez ts-node

```
npm install -g ts-node
```

- Exécutez le script TS

```
ts-node first-script.ts
```

Test unitaires

Pour exécuter des tests unitaires avec Mocha, une configuration supplémentaire est aussi nécessaire.

- Initialisez le package.json
- Définissez votre package.json comme ceci:

```
{
  "name": "samples",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "test": "mocha -r ts-node/register ./**/*.spec.ts"  ①
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "chai": "^4.1.2",
    "mocha": "^3.5.3",
    "ts-node": "^3.3.0",    ②
    "typescript": "^2.5.3"  ③
  },
  "dependencies": {
    "@types/chai": "^4.0.4",    ④
    "@types/mocha": "^2.2.43"   ⑤
  }
}
```

① : Il faut préciser à Mocha qu'il faut utiliser ts-node pour exécuter les tests

② : Ajout de la dépendance de développement ts-node

③ : Ajout de la dépendance de développement typescript

④ : Ajout de la dépendance @types/chai

⑤ : Ajout de la dépendance @types/mocha

1.3. Modéliser des données

1.3.1. Objet Film

- Créez un nouveau dossier et initialisez la configuration TypeScript
- Modélisez le fichier JSON suivant :

```
{
  "title"      : "12 Rounds 3: Lockdown",
  "releasedate": "Fri, 11 Sep 2015 00:00:00 -0700",
  "studio"     : "Lionsgate",
  "poster"     :
"http://trailers.apple.com/trailers/lions_gate/12rounds3lockdown/images/poster.jpg",
  "location"   : "/trailers/lions_gate/12rounds3lockdown/",
  "rating"     : "R",
  "genre"      : ["Action and Adventure"],
  "directors"  : "Stephen Reynolds",
  "actors"     : [
    "Dean Ambrose",
    "Roger Cross",
    "Daniel Cudmore",
    "Lochlyn Munro",
    "Ty Olsson",
    "Sarah Smyth"
  ],
  "trailers": [
    {
      "postdate" : "Mon, 27 Jul 2015 00:00:00 -0700",
      "url"      : "/trailers/lions_gate/12rounds3lockdown/",
      "type"     : "Trailer",
      "exclusive": false,
      "hd"       : true
    }
  ]
}
```

- Réalisez les tests unitaires avec mocha pour couvrir tous les fichiers que vous allez créer

1.3.2. Objet Formation

- Créez un objet Personne définit comme suit:
  - une personne a un prenom, nom de famille
  - une personne a un genre (Homme ou Femme)
  - une personne a une religion (paramètre privé et optionel) : Buddhism, Judaism, Islam, Catholicism, Atheist

    - une personne peut boire une boisson qui est définie par un nom, si elle contient de l'alcool et si elle est bio. La fonction boire logue les caractéristiques de la boisson.
    - une fonction technique toString qui affiche 'Prénom Nom'
- Créez un objet Formateur qui hérite d'une personne
    - un formateur possède une liste de connaissance (paramètre privé)
    - une connaissance est définie par un nom et un nombre d'années d'expérience
    - un formateur peut enseigner (fonction qui renvoi une liste de connaissances acquises par le formateur depuis plus de 2 ans)
- Créez un objet Stagiaire qui hérite d'une personne
    - un stagiaire possède une liste de certification
    - une certification est définie par un nom et une année d'obtiention
- Créez un objet Formation
    - une formation possède un nom, une date de début et de fin, une liste de stagiaires et une liste de formateur.
- Réalisez les tests unitaires avec mocha pour couvrir tous les fichiers que vous allez créer

Chapitre 2. La philosophie Angular

Voici la description de la philisophie faite par l'équipe Ninja Squad dans leur livre sur Angular qui est une référence en la matière.

Pour construire une application Angular, il faut saisir quelques trucs sur la philosophie du framework. Avant tout, Angular est un framework orienté composant. Vous allez écrire de petits composants, et assemblés, ils vont constituer une application complète. Un composant est un groupe d'éléments HTML, dans un template, dédiés à une tâche particulière. Pour cela, vous aurez probablement besoin d'un peu de logique métier derrière ce template, pour peupler les données, et réagir aux événements par exemple. Pour les vétérans d'AngularJS 1.x, c'est un peu comme le fameux duo "template / controlleur", ou une directive.

Il faut aussi dire qu'un standard a été défini autour de ces composants : le standard Web Component ("composant web"). Même s'il n'est pas encore complètement supporté dans les navigateurs, vous pouvez déjà construire des petits composants isolés, réutilisables dans différentes applications (ce vieux rêve de programmeur). Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de ReactJS, le framework tendance de Facebook ; EmberJS et AngularJS ont leur propre façon de faire quelque chose de similaire ; et les petits nouveaux Aurelia ou Vue.js parient aussi sur la construction de petits composants.

Angular n'est donc pas le seul sur le sujet, mais il est parmi les premiers (ou le premier ?) à considérer sérieusement l'intégration des Web Components (ceux du standard officiel). Mais écartons ce sujet, trop avancé pour le moment. Vos composants seront organisés de façon hiérarchique, comme le DOM : un composant racine aura des composants enfants, qui auront chacun des composants enfants, etc.

L'équipe Angular voulait aussi bénéficier d'une autre pépite du développement web moderne : ES6. Ainsi vous pouvez écrire tes composants en ES5 (pas cool !) ou en ES6 (super cool !). Mais cela ne leur suffisait pas, ils voulaient utiliser une fonctionnalité qui n'est pas encore standard : les décorateurs. Alors ils ont travaillé étroitement avec les équipes de transpileurs (Traceur et Babel) et l'équipe Microsoft du projet TypeScript, pour nous permettre d'utiliser des décorateurs dans nos applications Angular. Quelques décorateurs sont disponibles, permettant de déclarer facilement un composant et sa vue.

Chapitre 3. Angular: composants

3.1. Premier composant

3.1.1. Initialisation Typescript

- Créez un dossier et se placer dedans
- Créez le fichier de configuration de Typescript

```
    // Configuration de base pour un projet Angular
    tsc --init --target es5 --sourceMap --experimentalDecorators
--emitDecoratorMetadata
```

- Initialisez un projet node

```
    npm init -y
```

- Installez Angular et ses dépendances

```
    # Dependencies
    npm i --save @angular/core@"$NG" @angular/compiler@"$NG" @angular/common@"$NG"
@angular/platform-browser@"$NG" @angular/platform-browser-dynamic@"$NG" rxjs reflect-
metadata zone.js

    # Dev dependencies
    npm install --save-dev @types/core-js
```

3.1.2. Mon premier composant

- Créez un fichier app.component.ts
- Démarrer TypeScript

```
    tsc --watch --skipLibCheck
```

- Dans le fichier app.component.ts, créez et exportez une classe FormationAppComponent.

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

@Component({
    selector: 'formation-app',
    template: '<h1>Formation Angular</h1>'
})
export class FormationAppComponent {}
```

Angular utilise le décorateur @Component pour comprendre que c'est un composant. Ce décorateur requiert un objet de configuration avec a minima un selector. Selector indiquera à Angular ce qu'il faudra chercher dans nos pages HTML. A chaque fois que le sélecteur défini sera trouvé dans notre HTML, Angular remplacera l'élément sélectionné par notre composant.

3.1.3. Mon premier module

- Angular nécessite que les composants soient déclarés au niveau d'un module applicatif. Un module peut importer d'autres modules mais aussi exposer des composants et des services. Nous y reviendrons. Nous allons donc créer un premier module.
- Créez un fichier app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormationAppComponent } from './app.component';

@NgModule({
    imports: [BrowserModule],
    declarations: [FormationAppComponent], ①
    bootstrap: [FormationAppComponent]     ②
  })
export class AppModule { }
```

① Déclaration de notre composant. Tous les composants devront être ajoutés à ce tableau.

② Ce composant est le module racine de notre application. Bootstrap permet de le définir à Angular.

3.1.4. Lancer l'application

Nous devons enfin démarrer l'application !!!

- Créez un fichier main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

- Créez un fichier index.html.

Avec un projet angular.js 1.xxxx ou bien même un autre projet web, il suffit d'ajouter l'import du scrit angular dans la balise `head` de la page. Maintenant cela devient plus complexe, Angular est modulaire et chaque module ES6 peut être chargé dynamiquement. Cependant la notion de module n'existe pas en ES5 et les navigateurs n'ont pas encore implémentés cette spécification.

- Nous allons donc utilisé SystemJS pour charger notre application.

SystemJs est un chargeur de modules.

```
npm install --save systemjs
```

- Mettez à jour votre fichier HTML comme suit:

```
    <html>
        <head>
            <script src="node_modules/zone.js/dist/zone.js"></script>
            <script src="node_modules/reflect-metadata/Reflect.js"></script>
            <script src="node_modules/systemjs/dist/system.js"></script>
            <script>
              System.config({
                // the app will need the following dependencies
                map: {
                  '@angular/core': 'node_modules/@angular/core/bundles/core.umd.js',
                  '@angular/common':
'node_modules/@angular/common/bundles/common.umd.js',
                  '@angular/compiler':
'node_modules/@angular/compiler/bundles/compiler.umd.js',
                  '@angular/platform-browser': 'node_modules/@angular/platform-
browser/bundles/platform-browser.umd.js',
                  '@angular/platform-browser-dynamic':
'node_modules/@angular/platform-browser-dynamic/bundles/platform-browser-
dynamic.umd.js',
                  'rxjs': 'node_modules/rxjs'
                  },
                  packages: {
                    // we want to import our modules without writing '.js' at the end
                    // we declare them as packages and SystemJS will add the extension
for us
                    '.': {},
                    rxjs: {
                        main: 'index.js',
                        defaultExtension: 'js'
                    },
                    'rxjs/operators': {
                        'main': 'index.js',
                        'defaultExtension': 'js'
                    }
                  }
                }
              );
              // and to finish, let's boot the app!
              System.import('main');
            </script>
        </head>
        <body>
            <formation-app>
                Loading ...
            </formation-app>
        </body>
    </html>
```

- Installez un serveur web afin de démarrer l'application

```
npm install -g http-server
```

- Démarrez le serveur web

```
http-server
```

- Ouvrez votre navigateur à l'addresse http://localhost:8080

3.1.5. Application via Angular-cli

La mise en place d'une application Angular nécessite un nombre d'actions important. L'équipe Angular met donc à disposition un client Node.js afin d'initialiser un projet, avec un outillage complet (Tests, serveur de développement, gestion des environnements, ...). Ce client permet aussi de créer les nouveaux composants, injectables, ... Il permet de normaliser les architectures projets.

- Installez le module globale node @angular/cli

```
npm install -g @angular/cli
```

- Créez une nouvelle application formation-cli

```
ng new formation-cli
```

- Démarrez l'application

```
cd formation-cli
ng serve
```

`ng serve` permet de démarrer l'application en mode de développement. A chaque fois que vous modifierez un fichier le serveur détecte le changement et recharge la page automatiquement dans le navigateur.

- Ouvrez votre navigateur et allez à http://localhost:4200/
- Ajouter un composant

```
ng generate component formation
```

- Vous pouvez constatez que dans le dossier `src/app`, un dossier vient d'être créé. Ce dossier

contient 4 fichiers :

```
+- src
|   +- app
|   |   +- formation
|   |   |   +- formation.component.css  ①
|   |   |   +- formation.component.html ②
|   |   |   +- formation.component.spec.ts  ③
|   |   |   +- formation.component.ts   ④
|   |   ...
|   ....
....
```

① Style du composant

② Template HTML du composant

③ Fichier de test du composant

④ Composant (configuation, handler, cycle de vie)

Sources: 4. formation.component.ts

```
@Component({
  selector: 'fa-formation',
  templateUrl: './formation.component.html',
  styleUrls: ['./formation.component.css']
})
export class FormationComponent{
}
```

Les fichiers CSS et HTML du composant sont déclarés dans le décorateur Component.

- Vous pouvez aussi constaté que le composant a été ajouté automatiquement au module `app.module.ts`. Il est donc prêt à l'emploi. Vous pouvez maintent l'utiliser dans un template de votre application
- Nous allons donc utiliser le composant créé dans l'application. Pour cela, dans le composant (formation.component.ts), récupérer le selector défini, par exemple `app-formation`. Ajoutez dans le fichier `app.component.html` la balise `<app-formation></app-formation>`. Angular fait le reste pour nous. Lorsqu'il détecte cette balise `app-formation`, Angular activera le composant.

Le nom du selector n'est pas à choisir à la légère. En effet si vous ajoutez des librairies, il se peut qu'il y ait un nom qui soit identique. La bonne pratique est de définir un préfix à correspondant au client, au projet ou aux deux. Essayez de faire en forte qu'il soit le plus possible unique. Par exemple `maf-formation`, `maf` pour module angular formation.

- Vous devriez voir apparaitre dans l'interface votre composant.
- Dans le fichier `formation.component.html`, modifiez le texte, la modification devrait être affichée dès que vous sauvegardez le fichier

3.2. Interpolation

Je définirai l'interpolation comme une évaluation de code JavaScript dans un template HTML. Le code JavaScript qui peut être évalué est le code exposé dans le composant TypeScript qui définit le selector. Les propriétés publiques de ce composant seront accessibles dans le template.

Évaluer le JavaScript dans le template est possible via la syntaxte "double moustache" ou double accolade `{{}}`. Voici un d'un composant qui affiche la propriété `nom` dans son template.

Sources: 5. interpolate.component.html

```
<p>
  {{nom}}
</p>

<!--est équivalent -->

<p [textContent]="nom"></p>
```

Sources: 6. interpolate.component.ts

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

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

  nom: string = 'Bewizyu';

  constructor() { }

  ngOnInit() {
  }

}
```

Il est de fait possible d'interpoler tous les types de données.

Sources: 7. interpolate-extend.component.html

```
<div>
  <p>
    {{nom}} : {{formationAngular}}
  </p>

  <p>
    {{getTitle()}}
  </p>
  <p>
    Object: {{formation.nom}}
  </p>
</div>
```

Sources: 8. interpolate-extend.component.ts

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

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

  nom:string= 'Bewizyu'

  formationAngular:string= 'Module angular';

  formation:Formation = new Formation('Module Angular');

  getTitle() :string {
    return `Function:  ${this.nom} => ${this.formationAngular} `
  }

  constructor() { }

  ngOnInit() {
  }

}
```

- Créez un nouveau composant nommé `interpolate`
- Dans la classe TS, ajoutez une propriété `prenom` avec une valeur par défaut et interpolez la dans le template

- Ajoutez une propriété `nom` avec une valeur par défaut, dans le composant TS
- Ajoutez une propriété de type function `getFullName` qui renvoit la concaténation des propriétés `prenom` et `nom`
- Affichez la dans le template.
- Créez une classe `Formation` qui possède un nom et une description
- Dans le composant, ajoutez une propriété formation, instance de classe Formation et affichez le nom et la description dans le template

3.3. Paramètre optionnel

Il est possible qu'une variable n'est pas encore de valeur assignée lors de l'évaluation du template par Angular. Le paramètre optionnel nous permet d'afficher la variable uniqument lorsque celle-ci aura une valeur.

- Ajoutez une nouvelle propriété `formationJS` de type Formation sans valeur par défaut.

Sources: 9. interpolate-optional.component.ts

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

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

  nom:string= 'Bewizyu'

  formationAngular:string= 'Module angular';

  formation:Formation = new Formation('Module Angular');

  formationJS:Formation;

  getTitle() :string {
    return `Function:  ${this.nom} => ${this.formationAngular} `
  }

  constructor() { }


}
```

- Ajoutez l'interpolation dans le template

```
  <p>
    Optional: {{formationJS.nom}}
  </p>
```

- Vous devriez avoir l'erreur suivante

```
   Cannot read property 'nom' of undefined in [{{ formationJS.nom }} in
InterpolateOptionalComponent]
```

Angular nous met à disposition le paramètre optional `?` , appelé aussi 'Safe Navigation Operator'.

Il est fréquent qu'une donnée ne soit pas disponible à l'instanciation d'un composant. Un traitement asynchrone comme par exemple un appel réseau peut en être la cause. Avec le paramètre optionnel, cela se définira de la sorte:

```
<p>
  Optional: {{formationJS?.nom}}
</p>
```

- il nous reste à simuler un traitement asynchrone pour initialiser formationJS. Nous reviendrons plus loin sur la notion de cycle de vie d'un composant, mais sachez toutefois que la méthode `ngOnInit` est appelé qu'une seule fois à l'intialisation d'un composant après le constructeur.

Sources: 10. interpolate-optional.component.ts

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

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

  nom:string= 'Bewizyu'

  formationAngular:string= 'Module angular';

  formation:Formation = new Formation('Module Angular');

  formationJS:Formation;

  getTitle() :string {
    return `Function:  ${this.nom} => ${this.formationAngular} `
  }

  constructor() { }

  ngOnInit() {

    setTimeout(() => {
      this.formationJS= new Formation('Module JavaScript');
    }, 3000)
  }

}
```

3.4. Binding

3.4.1. Binding de propriété

Nous avons déjà abordé la partie interpolation qui permet d'afficher des données dynamiques dans nos templates. L'interpolation est en réalité un raccourci, nous pourrions aussi dire simplification du concept de binding.

Pour rappel voici les deux syntaxes pour faire une interpolation :

Sources: 11. interpolate.component.html

```
<p>
  {{nom}}
</p>

<!--est équivalent -->

<p [textContent]="nom"></p>
```

La version avec les {{}} est ce que l'on appelle l'interpolation comme je vous l'ai déjà présenté, la seconde avec les [] est en fait un binding.

Angular permet de faire du binding sur n'importe quelle propriété d'un élément du DOM.

En ajoutant les crochets [] autour d'une propriété vous rendez cette propriété dynamique.

Dans l'exemple ci-dessous, la propriété hidden d'un élément du DOM est rendue dynamique, la propriété isHidden du composant est évaluée pour savoir si on affiche ou non le composant.

Sources: 12. binding.component.html

```
<div [hidden]="isHidden">
  Binding hidden property
</div>
```

Sources: 13. binding.component.ts

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

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

  isHidden:boolean= true;

  constructor() { }

  ngOnInit() {
  }

}
```

Si maintenant, je veux l'afficher toutes les 2 secondes: .binding.component.ts

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

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

  isHidden:boolean= true;

  constructor() { }

  ngOnInit() {
    setInterval(() =>{
      this.isHidden = !this.isHidden;
    }, 2000)
  }

}
```

Le fonction `setInterval` permet de déclencher une fonction à intervalle régulier.

- En vous basant sur cet exemple, ajoutez dans le template HTML un texte statique avec une

balise `<p>`

- Changer la couleur de ce texte toutes les secondes, afin qu'il passe de vert à rouge. Vous pouvez le faire en utilisant la propriété `style.color`.

3.4.2. Binding événements

Avec l'interpolation et le binding de propriété, nous avons vu comment rendre l'interface dynanique en fonction des propriétés du composant déclaré en Typescript. Nous souhaitons aussi pouvoir déclencher des traitements en fonction d'événements déclenchés par le DOM ou par une action utilisateur, par exemple click, mouveover,ou tout autre événements disponible par l'API DOM.

Angular met à disposition la syntaxe suivante `()` pour binder un événement.

Sources: 14. evenement.component.html

```
<button (click)="handleClick()">Console log</button>
```

Sources: 15. evenement.component.ts

```
handleClick() {
  console.log('Le boutton a été cliqué');
}
```

Tous les événements disponibles dans une page HTML via l'API DOM sont accessibles. Pour binder l'événement dans un contexte Angular, la syntaxe est `(EVENT_NAME)`.

- Créez un bouton.
- Créez un texte mock dans une balise `p`
- Au click sur le bouton, masquer le texte si il est visible et rendez le visible si il est masqué.
- Loguez les événements `mouseover` et `mouseout` du texte.

3.5. Variable locale

Angular permet de déclarer des variables localement dans les templates HTML. Une variable locale représente l'élément du DOM sur laquelle elle est déclarée.

Sources: 16. variable.component.html

```
<div>
  <input type="text"
         #customVariable
         placeholder="Saisir une valeur">
  <div>test variable locale : {{customVariable.value}}</div>
</div>
```

La variable customVariable référence l'objet `HTMLInputElement`. Nous pouvons donc utiliser toutes les propriétés de cet objet.

Il faut toutefois attendre le prochain cycle de vie du composant qui déclenchera le checking du template pour que les changements soient pris en compte. Nous allons donc ajouter un bouton qui au click donne le focus à l'élément input représenté par la variable locale customVariable.

Sources: 17. variable.component.html

```
<div>
  <button (click)="customVariable.focus()">Focus the input</button>
</div>
```

Pour ainsi obtenir, au total:

Sources: 18. variable.component.html

```
<div>
  <input type="text"
         #customVariable
         placeholder="Saisir une valeur">
  <div>test variable locale : {{customVariable.value}}</div>
</div>

<div>
  <button (click)="customVariable.focus()">Focus the input</button>
</div>
```

Cette technique est régulièrement utilisée pour l'exécution d'une action sur un autre élément. Exactement comme dans l'exemple avec le focus.

3.6. Directives structurelles

Dans Angular, une directive ressemble à un composant, mais n'a pas de template. Cela sert à ajouter un comportement sur un élément du DOM. Les bindings d'évenements ou de propriétés ne permettent pas de modifier le DOM. Nous pourrions souhaiter ajouter ou supprimer un élément du DOM, par exemple.

Les 3 directives structurelles suivantes (ngIf, ngFor, ngSwitch) sont rendues disponibles par l'import du module `BrowserModule`. Si vous regardez le fichier `app.module.ts`, vous pouvez remarquer que ce module est importé par défaut dans une application Angular initialisée par le client angular-cli.

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [ ①
    BrowserModule ②
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

① Le décorateur NgModule permet d'importer les modules dont dépend le module AppModule

② Import du BrowserModule

3.6.1. ngIf

La directive `ngIf` permet d'ajouter ou de supprimer un élément du DOM en fonction d'une condition.

Sources: 19. if.component.html

```
<div *ngIf="isDisplay"><p>Formation Angular</p></div>
```

Sources: 20. if.component.ts

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

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

  isDisplay:boolean = true;

  constructor() { }

  ngOnInit() {
    setInterval(() => {
      this.isDisplay = !this.isDisplay;
    }, 1000)
  }

}
```

En fonction de la valeur de la propriété `isDisplay` l'élément sera ajouté ou supprimé. Il existe aussi une syntaxe un peu particulière pour effectuer un if else.

Sources: 21. if.component.html

```
  <div *ngIf="isDisplay; else title"><p>Formation Angular</p></div>
  <ng-template #title><p>Nom de la formation</p></ng-template>
```

Si la condition est fausse, cela affiche la l'élément qui déclare la variable local `title`. Vous pouvez remarquer aussi, l'ajout du composant `ng-template`. Je ne vais pas rentrer dans le détail lors de ce module de formation, mais sachez quand même que les directives structurelles (ngIf, ngFor, ..) sont en fait des composants du même type. Nous aurions pu écrire les exemples ci-dessus avec ce composant.

```
    <ng-template [ngIf]="isDisplay">
        <p>Formation Angular</p>
    </ng-template>
```

Pour approndir le sujet ⇒ documentation

- Affichez un titre et une description pour une formation
- Créez un bouton qui lorsqu'il est cliqué affiche ou masque le titre et la description.

3.6.2. ngFor

La directive `ngFor` permet d'instancier un template par élément d'une collection. Cette directive est très pratique dès lors que l'on souhaite afficher une liste d'éléments, par exemple une liste de formations.

Sources: 22. for.component.html

```
<ul>
  <li *ngFor="let formation of formations"> ①
    {{ formation.nom }}   ②
  </li>
</ul>
```

① Cette syntaxe particulière est appelée microsyntax. Dans l'exemple présent, cela permet de déclarer une variable `formation` qui représente un objet de la collection. Nous pouvons ensuite nous en servir dans le template.

② Utilisation de la variable `formation` pour afficher le nom avec une interpolation.

Sources: 23. for.component.ts

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

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

  formations:Array<Formation> = [];

  constructor() { }


  ngOnInit() {
    this.formations = [
      new Formation('Module Angular'),
      new Formation('Module JavaScript'),
      new Formation('Module TypeScript'),
    ]
  }
}
```

Angular met à disposition des variables en lien avec l'élément de la collection. Par exemple, l'index comme dans l'exemple ci-dessous.

Sources: 24. for.component.html

```
<ul>
  <li *ngFor="let formation of formations; index as i"> ①
    {{i}} - {{ formation.nom }}
  </li>
</ul>
```

① index as i est un alias

Ci-dessous la liste des variables qu'Angular met à disposition :

- `index`, index de l'élément dans la collection
- `even`, true si l'élément a un index pair
- `odd`, true si l'élément a un index impair
- `first`, true si l'élément est le premier de la collection
- `last`, true l'élément est le dernier de la collection

L'exemple ci-dessous, affiche en rouge le premier élément de la liste :

Sources: 25. for.component.html

```
<ul>
  <li *ngFor="let formation of formations; index as i; first as f"> ①
    <p [style.color]="getColorByElement(f)">{{i}} - {{ formation.nom }}</p>
  </li>
</ul>
```

① first as f est un alias

Sources: 26. for.component.ts

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

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

  formations:Array<Formation> = [];

  constructor() { }

  getColorByElement(isFirst:boolean = false) {
    return isFirst ? 'red' : 'black';
  }

  ngOnInit() {
    this.formations = [
      new Formation('Module Angular'),
      new Formation('Module JavaScript'),
      new Formation('Module TypeScript'),
    ]
  }
}
```

- Créez une liste de Formation dans le composant TS
- Afficher tous les noms des formations dans une liste en le préfixant avec son index dans la collection
- Le premier élément doit être affiché en rouge
- le dernier élément doit être affiché en vert
- Les éléments pairs afficheront un background gris

3.6.3. ngSwitch

Comme son nom l'indique cette directive permet de réalisé un swtich case dans le template

```
    <div [ngSwitch]="formations.length">
        <p *ngSwitchCase="0">Aucune formation disponibles</p>
        <p *ngSwitchCase="1">Une formation disponible : </p>
        <p *ngSwitchDefault>Formations disponibles :</p>
    </div>
```

- Ajoutez un titre avant la liste de formation en vous basant sur l’exemple précédent.

3.7. Style

Ces deux directives sont essentielles pour le templating.

3.7.1. ngStyle

Vous avez déja manipulé sans le savoir la directive `ngStyle`.

```
    <p [style.color]="'red'">Formation angular</p>
```

Si vous devez changer plusieurs style sur un élément, vous pouvez le déclarer ainsi :

```
<p [style.color]="'red'" [style.background]="'blue'">Formation angular</p>
```

Dans le cas où nous devons en modifier plusieurs nous aurions tendance à préférer cette syntaxe :

```
    <p [ngStyle]="{color: 'red', background: 'blue'}">Formation angular</p>
```

Comme dans dans une application sans Angular, nous privilégierons le style avec une classe CSS plutôt que de définir le style inline.

3.7.2. ngClass

La directive `ngClass` fonctionne exactement comme `ngStyle` mais sur les classes CSS.

Sources: 27. ngstyle.component.css

```
.sapin {
  color: red;
  background: blue;
}

.sapin-italic {
  font-style: italic;
}
```

Sources: 28. ngstyle.component.html

```
  <p [class.sapin]="true" [class.sapin-italic]="true">Formation angular</p>
```

Dans le cas où nous devons en modifier plusieurs nous utiliserons cette syntaxe :

Sources: 29. ngstyle.component.html

```
<p [ngClass]="{'sapin': true, 'sapin-italic': true}">Formation angular</p>
```

- Reprenez le TP de la directive ngFor qui mettait le premier élément en rouge, ...
- Remplacez toutes les styles réalisés avec la directive `ngStyle` par `ngClass`

3.8. Pipe

Un pipe permet de formater les données pour l'affichage, un exemple courant est la date. Il nous ait souvent demandé d'afficher une date de la sorte '19/11/2014' ou bien '19 novembre 2014'.

Angular nous met à disposition toute une série de pipe pour formater notre affichage. Nous indiquerons lors de l'interpolation le caractère | après la donnée, suivi du pipe que nous souhaitons appliqué. Il est également possible de chainer plusieurs pipes.

Voici la liste des pipes fournis par Angular:

- `json`: affiche les données au format JSON

```
<p>{{ formation | json }}</p>

<!-- produira -->

<p>{nom: 'Module Angular', description: '.....'}</p>
```

- `slice` permet d'afficher que les caractères ou les éléments d'une collection compris entre 2 index.

```
<!-- formation.nom = 'module' -->

<p>{{ formation.nom | slice:0:2 }}</p>

<!-- produira -->

<p>mod</p>
```

- `uppercase`: Affiche les données en majuscule
- `lowercase`: Affiche les données en minuscule
- `titlecase`: Affiche les données avec une lettre majucule au début de chaque mot
- `number`: Affiche un nombre formaté (nombre de chiffre après la virgule, ...)
- `percent`: Affiche un pourcentage
- `currency`: Affiche un nombre dans la devise souhaitée par le code ISO de la devise ('EUR', 'USD'...)

```
<p>{{ 7.4 | currency:'EUR' }}</p>
<!-- affiche 'EUR7,4' -->

<p>{{ 7.4 | currency:'USD':true }}</p>
<!-- affiche '7.4' -->
```

- `date`: permet de formater une date

```
  <p>{{ birthday | date:'dd/MM/yyyy' }}</p>
  <!-- affiche '19/11/2014' -->

  <p>{{ birthday | date:'longDate' }}</p>
  <!-- will display 'November 14, 2014' -->
```

- `async`: permet d'afficher des données obtenues de manière asynchrone. Un pipe async retourne une chaîne de caractères vide jusqu'à ce que les données deviennent disponibles. Vous pouvez remarquer dans l'exemple suivant que `asyncPromise` est une propriété de type Promise. Les données affichées sont celles transmises dans le resolve de la promesse.

Sources: 30. pipes.component.html

```
<p>{{asyncPromise | async}}</p>
```

Sources: 31. pipes.component.ts

```
asyncPromise: any = new Promise(resolve => {
  setTimeout(() => resolve('Async data'), 1500);
});
```

- `keyvalue` : Permet d'itérer sur les propriétés d'un objet ou une Map.

Sources: 32. pipes.component.html

```
<ul>
  <li *ngFor="let entry of keyValueObj | keyvalue">
    {{ entry.key }} - {{ entry.value }}
  </li>
  <!-- itère sur les propriétés de l'objet -->
  <!--<li>key1 - value1</li>-->
  <!--<li>key2 - value2</li>-->
</ul>`
```

Sources: 33. pipes.component.ts

```
keyValueObj : {key1 : string, key2 : string} = {
  key1 : 'Value 1',
  key2 : 'Value 2',
}
```

- Reprenez le TP de la directive ngFor qui mettait le premier élément en rouge, ...
- Ajouter un prix à une formation, ainsi qu'une date de début et une date de fin

- Pour chaque formation, affichez le titre en 'titlecase', le prix en euro et les dates de début et de fin sous le format '19/11/2014'.

3.9. Décorateur @Input

Depuis le début du TP nous, manipulons tout notre code dans le même composant. Il est temps de passer en mode normal ;). Angular propose une approche 'component first', ce qui signifie qu'il faut privilégier une approche orienté composant. Il est une bonne pratique de garder des composants aussi simple que possible. C'est à dire que nous allons réaliser plein de petits composants et créer ainsi une arborescence de composants.

- Créez un nouveau composant `FormationItem`. Ce composant va représenter le bloc d'éléments sur lequel nous itérons avec la directive `ngFor`.

Pour rappel, créez les composants avec `angular-cli`

```
ng generate component formation-item
```

- Mettez à jour le template du composant `FormationItem` avec le bloc d'élément sur lequel vous itérez dans le ngFor. Vous devriez avoir quelque chose comme suit, avec les styles et les pipes en plus.

Sources: 34. formation-item.component.html

```
<div>
  <h4>{{formation?.nom}}</h4>
  <p>{{formation?.description}}</p>
  <p>{{formation?.prix}}</p>
</div>
```

- Déclarez composant `formation-item` dans la boucle ngFor

Sources: 35. playground.component.html

```
  <ul>
    <li *ngFor="let formation of formations">
      <app-formation-item></app-formation-item>
    </li>
  </ul>
```

Normalement dans la console de votre navigateur, vous devez constatez une erreur.

```
compiler.es5.js:1694 Uncaught Error: Template parse errors:
Can't bind to 'formation' since it isn't a known property of 'app-formation-item'.
```

Dans le template de ce composant, des interpolations sont effectuées sur la propriété formation. Actuellement vous ne transmettez pas l'objet formation à votre composant.

Le concept de binding de propriétés que nous avons vu précédemment, va nous permettre de répondre à cette problématique. En utilisant la syntaxe de binding de propriété avec les crochets [], nous 'transmettons' au composant enfant des données.

- Mettez à jour votre code en prenant exemple que le code suivant :

Sources: 36. playground.component.html

```
<ul>
  <li *ngFor="let formation of formations">
    <app-formation-item [formation]="formation"></app-formation-item>
  </li>
</ul>
```

Cependant votre application est encore en erreur, vous pouvez le vérifier dans votre console. En effet, nous essayons de transmettre l'objet formation vers une propriété formation du composant formation-item. Mais nous ne l'avons pas encore déclarée.

Le binding de propriété est le moyen de transmettre des données d'un composant parent vers un composant enfant. Le composant enfant doit déclarer la propriété comme une entrée de données. Ce que nous pouvons faire avec le décorateur `@Input` fourni par le module `@angular/core`.

Sources: 37. formation-item.component.ts

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

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

  @Input()
  formation:Formation;

  constructor() { }

  ngOnInit() {
  }

}
```

- Réalisez les adaptations nécessaires dans le composant formation-item.

3.10. Décorateur @Output

De la même manière que nous voulons passer des données d'un composant enfant à un composant parent, nous souhaitons aussi qu'un composant puisse notifier, envoyer des données à son parent. La sortie d'un composant vers son parent se réalise toujours via des événements et l'objet `EventEmitter`. La déclaration de sortie s'effectue via le décorateur `@Output`.

Sources: 38. formation-item-output.component.ts

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
import Formation from '../model/Formation';

@Component({
  selector: 'app-formation-item-output',
  templateUrl: './formation-item-output.component.html',
  styleUrls: ['./formation-item-output.component.css']
})
export class FormationItemOutputComponent implements OnInit {

  @Input()
  formation:Formation;

  @Output()
  formationSelected:EventEmitter<Formation> = new EventEmitter<Formation>();

  selectFormation() {
    this.formationSelected.emit(this.formation);
  }

  constructor() { }

  ngOnInit() {
  }

}
```

Vous pouvez constater qu'une propriété de sortie (@Output) `formationSelected` est déclarée. Une fonction `selectFormation` est aussi créé, elle émet un événement avec une donnée, dans ce cas, une formation. Il reste donc à déclencher cette fonction au click du titre de la formation, comme ci-dessous :

Sources: 39. formation-item-output.component.html

```
<div>
  <h4 (click)="selectFormation()">{{formation?.nom}}</h4>
  <p>{{formation?.description}}</p>
  <p>{{formation?.prix}}</p>
</div>
```

Dans le composant parent, nous pouvons binder une propriété de type fonction sur l'événement `formationSelected`, comme nous l'avons vu lors du binding d'événements.

Sources: 40. playground.component.html

```
  <ul>
    <li *ngFor="let formation of formations">
      <app-formation-item-output
        [formation]="formation"
        (formationSelected)="handleFormationSelected($event)">
      </app-formation-item-output>
    </li>
  </ul>
```

Sources: 41. playground.component.ts

```
  handleFormationSelected(formation) {
    console.log('Formation selected', formation);
  }
```

- Réalisez le même traitement dans votre application. Au click sur le titre de la formation, déclenchez un événement qui sera capté dans le composant parent afin d'afficher la formation sélectionnée dans une popin modale (alert)

3.11. Cycle de vie d'un composant

Un composant possède un cycle de vie. Depuis le début des TP's, vous en utilisez déjà une d'alleurs, `ngOnInit`. Angular met plusieurs phases à disposition.

- Phases courantes :
  - `ngOnChanges` est appelée quand la valeur d'une propriété bindée est modifiée. Elle recevra une map changes, contenant les valeurs courantes et précédentes du binding.
  - `ngOnInit` sera appelée une seule fois après le premier changement
  - `ngOnDestroy` est appelée quand le composant est supprimé.
- Phases avancées :
  - `ngDoCheck` est légèrement différente. Si elle est présente, elle sera appelée à chaque cycle de détection de changements, redéfinissant l'algorithme par défaut de détection, qui inspecte les différences pour chaque valeur de propriété bindée. Cela signifie que si une propriété au moins est modifiée, le composant est considéré modifié par défaut, et ses enfants seront inspectés et réaffichés.
  - `ngAfterContentInit` est appelée quand tous les bindings du composant ont été vérifiés pour la première fois.
  - `ngAfterContentChecked` est appelée quand tous les bindings du composant ont été vérifiés, même s'ils n'ont pas changé.
  - `ngAfterViewInit` est appelée quand tous les bindings des directives enfants ont été vérifiés pour la première fois.
  - `ngAfterViewChecked` est appelée quand tous les bindings des directives enfants ont été vérifiés, même s'ils n'ont pas changé. Cela peut être utile si ton composant attend quelque chose de ses composants enfants. Comme ngAfterViewInit, cela n'a de sens que pour un composant (une directive n'a pas `de vue).

Voici un exemple de détection de changement via `ngOnChanges`

Sources: 42. lifecycle.component.ts

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

const NOM_ANGULAR : string = 'Module Angular';
const NOM_JS : string = 'Module JavaScript';

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

  formation:Formation = new Formation(NOM_ANGULAR);

  constructor() { }

  ngOnInit() {
    setInterval(()=> {
      this.formation = new Formation(this.formation.nom === NOM_ANGULAR ? NOM_JS :
NOM_ANGULAR);
    }, 1000)
  }

}
```

Sources: 43. lifecycle.component.html

```
<div>
  <app-lifecycle-onchange [formation]="formation"></app-lifecycle-onchange>
</div>
```

Sources: 44. lifecycle-onchange.component.ts

```
import { Component, OnInit, Input, OnChanges, SimpleChanges } from '@angular/core';
import Formation from '../model/Formation';

@Component({
  selector: 'app-lifecycle-onchange',
  templateUrl: './lifecycle-onchange.component.html',
  styleUrls: ['./lifecycle-onchange.component.css']
})
export class LifecycleOnchangeComponent implements OnInit, OnChanges {

  @Input()
  formation:Formation;

  constructor() { }

  ngOnInit() {
  }

  ngOnChanges(change:SimpleChanges) {
    const f = change['formation'];
    if (f.previousValue) {
      console.log(`Previous : ${f.previousValue.nom}`);
    }
    console.log(`Current : ${f.currentValue.nom}`);
  }

}
```

Sources: 45. lifecycle.component.html

```
<div>
  <h4>{{formation?.nom}}</h4>
  <p>{{formation?.description}}</p>
  <p>{{formation?.prix}}</p>
</div>
```

3.12. TODO application

- Créez une nouvelle application.
- Créez une classe modèle pour un Todo. Un Todo est défini comme ci-dessous.

```
{
  "title": "Send a mail",
  "isDone":false
}
```

- Générez un composant TodoList qui affichera une liste de TodoItem et le nombre total de Todos présents dans la liste

La bonne pratique est de créer un composant TodoContainer qui aura la responsabilité de manager la liste des Todos. Il pourra ainsi transmettre cette liste à son composant enfant TodoList.

- Créez une liste de todos mockés dans un premier temps (4 ou 5)
- Générez un composant TodoItem qui affiche le titre en uppercase et une checkbox bindée sur le boolean isDone d'un Todo.
- Quand le todo est effectué, l'utilisateur coche la checkbox, ce qui a pour effet de barrer le titre du Todo.
- Générez un composant TodoForm qui permet de saisir un Todo et le positionner au dessus du composant TodoList dans l'interface. Il sera composé d'un champ de saisie pour le titre et d'un bouton d'ajout. Au click sur le bouton d'ajout, le todo est ajouté à la liste.

A ce stade, nous n'avons pas encore abordé la notion de formulaire ni le binding bi-directionnel. Par contre, nous avons déja abordé le binding de propriétés et d'événement. Voici un exemple que code, qui vous permettra d'avancer.

Sources: 46. test-checkbox.component.html

```
<div>
  <label for="mc">Sample checkbox</label>
  <input
    type="checkbox"
    name="mc"
    id="mc"
    [checked]="todo.isDone"
    (change)="handleCheckBoxChange()"
  >

  <p>Valeur de la checkbox: {{todo.isDone}} </p>

  <button (click)="handleClick()">Click me</button>
</div>
```

Sources: 47. test-checkbox.component.ts

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

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

  todo: {title:string, isDone:boolean} = {
    title: 'Finir le TP ;)',
    isDone: false
  }

  constructor() { }

  handleCheckBoxChange() {
    this.todo.isDone = !this.todo.isDone;
  }

  handleClick() {
    console.log("TODO", this.todo);
  }

  ngOnInit() {
  }

}
```

Le composant TodoForm doit transmettre le Todo à son composant parent pour l'ajouter à la liste

- Supprimez la liste mockée maintenant que l'on peut ajouter des todos.
- Quand la liste est vide un texte doit être affiché pour informer l'utilisateur qu'il n'a aucun todo. "Vous n'avez aucun todo"
- Dans le composant TodoForm, ajoutez un bouton reset qui réinitialise la liste des todos

Chapitre 4. Angular: Injections de dépendances

4.1. Concept

Tout d'abord, définissons ce que l'on appelle une dépendance. Il est une bonne pratique lorsque nous développons des applications qu'elles soient web ou java ou autres, d'architecturer notre code en utilisant des design patterns. Ces design patterns vont nous conduire à séparer notre code en classes, services, factories, composants, ... Cela doit nous permettre d'avoir des 'bouts de codes' réutilisables, avec une responsabilité claire et définie et ainsi éviter la duplication de code ou bien même faciliter la maintenance de notre application.

Par exemple, un composant dépendra régulièrement d'un service métier que ce soit pour récupérer des données sur un serveur, ou factoriser un traitement qui sera utile dans d'autres composants. Ce service est ce que l'on nomme une dépendance.

Si nous poursuivons avec l'exemple précédent, le composant pourrait instancier le service ou bien même récupérer l'instance d'un singleton.

Plutôt que de laisser le composant instancier le service, c'est Angular (le framework) qui va avoir la responsabilité de fournir au composant la dépendance. C'est technique se nomme injection de dépendance ou encore l'inversion de contrôle (IOC).

Il sera d'autant plus aisé de tester notre composant mais nous pourront aussi configurer quelle sera l'implémentation de cette dépendance.

4.2. Première dépendance

Une dépendance est en réalité une simple classe.

- Créez un nouveau projet

```
ng new dependencies-injection
```

- Créez un composant `formation-list`
- Créez une classe Formation avec une propriété nom et description
- Dans le composant `formation-list.component.ts`, déclarez une propriété formations qui est une liste de formation.
- Dans le template du composant formation-list, affichez la liste de formations (ngFor)
- Créez dossier `src/app/services`
- Créez une classe `FormationService` qui déclare une méthode `getFormations` retournant une liste de formations.
- Dans ce service, getFormations retourne une liste de formation mockée

Voici le service que vous devriez obtenir :

Sources: 48. formationServices.ts

```
import Formation from '../../models/Formation';

export default class FormationService {

  getFormation(): Array<Formation> {
    return[
      new Formation('Module Angular'),
      new Formation('Module JavaScript'),
      new Formation('Module TypeScript'),
    ];
  }
}
```

Il nous reste maintenant à injecter cette dépendance dans le composant formation-list. Pour cela, il suffit de le déclarer dans le constructeur du composant formation-list. Lors de l'initialisation du composant récupèrez la liste de formations depuis le service.

Sources: 49. formation-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import Formation from "../models/Formation";
import FormationService from "../services/formations/formation.services";

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

  formations: Array<Formation>

  constructor(private formationService:FormationService) { }

  ngOnInit() {
    this.formations = this.formationService.getFormation();
  }

}
```

A ce stade, vous devriez obtenir dans la console du navigateur l'erreur suivante :

```
    ERROR Error: No provider for FormationService!
```

Pour résumer, nous avons créé un service qui a été injecté dans le composant formation-list. Ce service doit être injecté par Angular dans le composant, mais nous ne lui avons pas encore précisé ce qu'il devait faire, ce qu'il nous indique dans l'erreur.

Pour corriger cette erreur, nous devons lui déclarer ce provider. Nous pouvons le faire dans le module de notre composant.

Sources: 50. app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { FormationListComponent } from './formation-list/formation-list.component';
import FormationService from "./services/formations/formation.services";

@NgModule({
  declarations: [
    AppComponent,
    FormationListComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [
    FormationService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Vous venez de créez votre premier service.

4.3. Providers

Nous venons de créer un premier service. La dernière étape a été la déclaration du service dans le tableau de providers du module. Comme je vous l'ai déjà mentionné, Angular manage les dépendances à injecter dans les composants, services (nous reviendrons la dessus un peu plus tard).

L'ajout majeur d'Angular est la modularité de déclaration de ses dépendances. Pour chaque provider défini, Angular a besoin, d'un token / clé (nom de la dépendance) et de la classe à instancier. Angular instanciera l'objet dès qu'un composant requiert la dépendance. Il conservera l'intance et la fournira aux autres composant qui la demanderont.

Un provider est un singleton.

Pour illustrer les différentes configurations possibles de providers, voici un exemple:

Sources: 51. app.module.provider.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { FormationListComponent } from './formation-list/formation-list.component';
import FormationService from "./services/formations/formation.services";
import MockFormationService from "./services/formations/mock-formation.services";

const condition:boolean = true;

@NgModule({
  declarations: [
    AppComponent,
    FormationListComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [
    FormationService,  ①
    { provide: FormationService, useClass: FormationService }, ②
    { provide: 'FormationServiceCustomName', useClass: FormationService }, ③
    { provide: FormationService, useClass: MockFormationService }, ④
    {
      provide: FormationService,  ⑤
      useFactory: () => condition ? new FormationService() : new
MockFormationService()
    }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

① Syntaxe courte, le token est identique à la classe

② Syntaxe longue, le token est identique à la classe

③ Syntaxe longue, le token est différent de la classe

④ Syntaxe longue, la classe à implémenter est différente du token, très pratique pour des tests ou pour mocker un service pendant le développement

⑤ factory, il est également possible de fournir une classe à implementer qui sera défini en fonction d'un besoin spécifique.

Tous les providers déclarés dans le module seront injectables dans des services ou des composants du module. Il est toutefois possible de déclarer des providers au niveau des composants dans le décorateur `@Component`.

Sources: 52. formation-list-provider.component.ts

```
import { Component, OnInit } from '@angular/core';
import Formation from "../models/Formation";
import FormationService from "../services/formations/formation.services";

@Component({
  selector: 'app-formation-list',
  templateUrl: './formation-list.component.html',
  styleUrls: ['./formation-list.component.css'],
  providers: [
    FormationService
  ]
})
export class FormationListComponent implements OnInit {

  formations: Array<Formation>

  constructor(private formationService:FormationService) { }

  ngOnInit() {
    this.formations = this.formationService.getFormation();
  }

}
```

Le dernier point d'attention concernant les providers est important à comprendre. Chaque composant va donc avoir son propre injecteur. Si une dépendance n'est pas résolue dans cet injecteur, il va la chercher dans son composant parent et ainsi de suite jusqu'aux providers du module. Si il n'est pas présent dans tous ces injecteurs alors Angular lève une erreur.

- Pour l'instant, notre service FormationService n'est utilisé que par le composant formation-list. Déclarez le provider dans le composant pluôt que dans le module.

4.4. @Injectable

En vous expliquant les providers, j'ai mentionné que des services pouvaient être injectés dans d'autres services. Le mécanisme est identique à la déclaration d'un service. La seule différence est qu'il faut indiquer à Angular que le service dans lequel nous souhaitons injecté une dépendance est éligible pour l'injection. Ceci peut être réalisé par le décorateur `@injectble`.

Dans le même temps, nous allons rendre nos traitements asynchrone, car pour l'instant tout est synchrone :(

- Créez un service `FormationApi` qui retourne une promise. Quand elle est résolu au bout de 2 secondes (setTimeout), elle retourne le tableau de formations mocké.

Sources: 53. formation.api.ts

```
import Formation from "../../models/Formation";

export default class FormationApi {
  fetchFormation():Promise<Array<Formation>> {
    return new Promise((resolve)=> {
      setTimeout(()=> {
        resolve([
          new Formation('Module Angular'),
          new Formation('Module JavaScript'),
          new Formation('Module TypeScript'),
        ])
      }, 2000);
    });
  }
}
```

- Dans le services FormationServices, récupérer les données en appelant le services FormationApi.
- Déclarez l'import du services FormationApi dans le constructeur du service et décorez la classe avec le décorateur `@Injectable` qui est mis à disposition par `@angular/core`

Sources: 54. injectable-formation.services.ts

```
import {Injectable} from '@angular/core';
import Formation from '../../models/Formation';
import FormationApi from './formation.api';

@Injectable()
export default class FormationService {

  constructor(private formationApi: FormationApi){}

  getFormation(): Promise<Array<Formation>> {
    return this.formationApi.fetchFormation();
  }
}
```

- Déclarez le provider FormationApi dans le module

Sources: 55. app.module.injectable.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';

import {AppComponent} from './app.component';
import {FormationListComponent} from './formation-list/formation-list.component';
import FormationService from "./services/formations/formation.services";
import FormationApi from "./services/formations/formation.api";

@NgModule({
  declarations: [
    AppComponent,
    FormationListComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [
    FormationApi
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Depuis la version de Angular 6.0, il est possible d'enregister le provider directement dans le décorator Injectable via la propriété `provideIn`. Cette approche est même devenue la déclaration recommandée pour définir un provider. Vous n'aurez plus besoin de réaliser la déclaration dans le décorateur NgModule.

```
import { Injectable } from '@angular/core';
@Injectable({
    providedIn: 'root'
})
export class ApiService {

    get(path) {
    // todo: call the backend API
    }
}
```

La valeur `root` n'est pas la seule valeur possible. Root représente l'injecteur root d'une application Angular. Il est également possible de définir comme valeur le nom d'un module Angular.

```
import { Injectable } from '@angular/core';
@Injectable({
    providedIn: 'MonModuleCustom'
})
export class ApiService {

    get(path) {
    // todo: call the backend API
    }
}
```

4.5. TODO application

- Reprenez le projet TODO application
- Créez un service `TodoServices`
- Décaler tous les traitements d'ajout, suppression de Todo dans le service TodoServices

La liste des todos devra être gérée dans ce service.

Chapitre 5. Angular: HttpClientModule

5.1. HttpClientModule

Lors du chapitre sur les directives structurelles, je vous ai déja parlé de la notion de module, sans entrer dans le détail. Angular est un framework modulaire, certains modules sont externalisés dans des dépendances Node.js que nous pouvons installer en dépendance de notre application via npm.

Voici les dépendances installé par défaut dans un projet Angular instancié via le client angular-cli

```
    "@angular/animations": "^6.1.0",
    "@angular/common": "^6.1.0",
    "@angular/compiler": "^6.1.0",
    "@angular/core": "^6.1.0",
    "@angular/forms": "^6.1.0",
    "@angular/http": "^6.1.0",
    "@angular/platform-browser": "^6.1.0",
    "@angular/platform-browser-dynamic": "^6.1.0",
    "@angular/router": "^6.1.0",
    "core-js": "^2.5.4",
    "rxjs": "~6.2.0",
    "zone.js": "~0.8.26"
  },
  "devDependencies": {
```

La raison est simple, chaque application n'a pas le même besoin, et nous pouvons inclure les dépendances nécessaires à chaque projet. Nous évitons ainsi d'embarquer des fonctionnalités inutiles pour optimiser notre application web, le temps d'affichage sera optimisé et l'expérience utilisateur plus fluide.

HttpClientModule ou HttpModule

Pour utiliser les API fournies par Angular pour forger des appels réseaux sur le protocole HTTP, l'application doit avoir la dépendance Node.js `@angular/common/http`. Cette dépendance nous met à disposition le module `HttpClientModule`.

Vous pouvez remarquer aussi la dépendance '@angular/http'. Cette dépendance fournit le module `HttpModule`. Ce module a complètement été réécrit au profit du `HttpClientModule`, qu'il faut prévilégier aujourd'hui.

5.1.1. HttpClient

- Pour utiliser le service HttpClient, il nous faut en premier lieu importer le module `HttpClientModule` dans le module de l'application angular

Sources: 56. app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule }  from '@angular/common/http';

import { AppComponent } from './app.component';
import {GithubApiService} from "./services/api/github.api.service";


@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [
    GithubApiService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

- Créez un service githubApi

```
    ng generate service services/githubApi
```

- Injectez le service `HttpClient`

Sources: 57. github.api.service.ts

```
  constructor(private http: HttpClient) { }
```

- Créez une fonction `getUser` qui utilise la méthode `get` du service HttpClient. Comme son nom l'indique, elle permet de faire un appel HTTP avec le verbe GET.

Sources: 58. github.api.service.ts

```
const API_BASE_URL: string = 'https://api.github.com/';
const API_USERS: string = 'users/';

  getUser(login:string) {
    return this.http.get(`${API_BASE_URL}${API_USERS}${login}`);
  }
```

Voici comment récupérer les données retournées par l'appel réseau:

Sources: 59. github.api.service.spec.ts

```
    service.getUser('nartawak')
      .subscribe((result:any) => {
        console.log('next');
        expect(result.login).toBe('nartawak')
      }, () => {
        console.log('error');
        fail("Do not fail")
      }, () => {
        console.log('complete');
      });
```

Vous pouvez remarquer, la méthode `subscribe` qui est appelée pour récupérer les données. Nous reviendrons sur cette notion un peu plus loin. Néanmoins, sachez quand même que la fonction `http.get` (comme tous les verbes HTTP) retourne ce que l'on nomme un `Observable`, qui possède une propriété `subscribe`.

Dans l'exemple, nous avons fait un appel réseau HTTP GET mais tous les verbes HTTP sont bien sûr disponible.

Ci dessous la liste exhaustive:

- get
- post
- put
- delete
- patch
- head
- jsonp

5.1.2. Paramètres

Il est possible de déclarer des `query parameters` sur une requête :

Sources: 60. params.service.ts

```
import { Injectable } from '@angular/core';
import {HttpClient, HttpParams} from "@angular/common/http";

@Injectable()
export class ParamsService {

  constructor(private http: HttpClient) { }

  sampleParams() {
    const params = new HttpParams()
      .set('login', 'nartawak');

    this.http.get(`http://sample.com`, { params })
      .subscribe(() => {});

    // => Crée un appel HTTP GET http://sample.com?login=nartawak
  }

}
```

5.1.3. Headers

De même, voici un exemple pour ajouter des headers sur une requête :

Sources: 61. headers.service.ts

```
import {Injectable} from '@angular/core';
import {HttpClient, HttpHeaders} from "@angular/common/http";

@Injectable()
export class HeadersService {

  constructor(private http: HttpClient) { }

  sampleHeader() {
    const headers = new HttpHeaders()
      .set('Content-Type', 'application/json');

    this.http.get(`http://sample.com`, { headers })
      .subscribe(() => {});

  }

}
```

5.1.4. Interceptors

Comme le nom l'indique, un interceptor sert à intercepter les requêtes ou les réponses des appels réseaux. Cela s'avère particulièrement utile pour mutualiser un traitement sur toutes les requêtes, comme par exemple ajouter un header, ou faire une gestion d'erreur centralisée.

Sources: 62. samples-header-interceptor.service.ts

```
import { Injectable } from '@angular/core';
import {HttpEvent, HttpHandler, HttpInterceptor, HttpRequest} from
"@angular/common/http";
import {Observable} from "rxjs/Observable";

@Injectable()
export class SamplesHeaderInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    // Le traitement est déclenché si la requête sur l'API sample.com
    if (req.url.includes('sample.com')) {
      // Ajout du header Content-Type
      // => Il est important de noter qu'une requête est immuable, vous devez la
cloner
      const clone = req.clone({ setHeaders: { 'Content-Type': 'application/json'}});

      return next.handle(clone);
    }

    return next.handle(req);
  }
}
```

Sources: 63. samples-error-interceptor.service.ts

```
import {Injectable} from '@angular/core';
import {HttpErrorResponse, HttpEvent, HttpHandler, HttpInterceptor, HttpRequest} from
"@angular/common/http";
import {Observable, throwError} from "rxjs";
import {catchError} from "rxjs/operators";

@Injectable()
export class SamplesErrorInterceptor implements HttpInterceptor {
  constructor() {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      catchError((errorResponse: HttpErrorResponse) => {
        if (errorResponse.status === 404) {
          // TODO: Gérer l'erreur
        }

        return throwError(errorResponse);
      }));
  }
}
```

Les intercepteurs étant des services, nous devons les déclarer dans le module en tant que providers. Ils sont ajoutés au tableau 'HTTP_INTERCEPTORS'.

Sources: 64. app.module.interceptors.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {HTTP_INTERCEPTORS, HttpClientModule} from '@angular/common/http';

import {AppComponent} from './app.component';
import {GithubApiService} from "./services/api/github.api.service";
import {SamplesHeaderInterceptor} from "./services/interceptors/samples-header-
interceptor.service";
import {SamplesErrorInterceptor} from "./services/interceptors/samples-error-
interceptor.service";


@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [
    GithubApiService,
    { provide: HTTP_INTERCEPTORS, useClass: SamplesHeaderInterceptor, multi: true },
    { provide: HTTP_INTERCEPTORS, useClass: SamplesErrorInterceptor, multi: true }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

5.2. Observable kesako ?

Voici la description de la programmation réactive faite par l'équipe Ninja Squad dans leur livre sur Angular.

La programmation réactive n'est pas quelque chose de fondamentalement nouveau. C'est une façon de construire une application avec des événements, et d'y réagir (d'où le nom). Les événements peuvent être combinés, filtrés, groupés, etc. en utilisant des fonctions comme map, filter, etc. C'est pourquoi vous croiseraez parfois le terme de "programmation fonctionnelle réactive" (functional reactive programming). Mais pour être tout à fait précis, la programmation réactive n'est pas foncièrement fonctionnelle, parce qu'elle n'inclue pas forcément les concepts d'immuabilité, l'absence d'effets de bord.

Dans la programmation réactive, toute donnée entrante sera dans un flux. Ces flux peuvent être écoutés, évidemment modifiés (filtrés, fusionnés, ...), et même devenir un nouveau flux que l'on pourra aussi écouter. Cette technique permet d'obtenir des programmes faiblement couplés : vous n'avez pas à vous soucier des conséquences de votre appel de méthode, vous vous contenterez de déclencher un événement, et toutes les parties de l'application intéressées réagiront en conséquence. Et peut-être même qu'une de ces parties va aussi déclencher un événement.

Et bien Angular est construit sur de la programmation réactive, et nous utiliserons aussi cette technique pour certaines parties. Répondre à une requête HTTP ? Programmation réactive. Lever un événement spécifique dans un de nos composants ? Programmation réactive. Gérer un changement de valeurs dans un de nos formulaires ? Programmation réactive.

Dans la programmation réactive, tout est un flux. Un flux est une séquence ordonnée d'événements. Ces événements représentent des valeurs , des erreurs, ou des terminaisons. Tous ces événements sont poussés par un producteur de données, vers un consommateur. En tant que développeur, votre job sera de vous abonner (subscribe) à ces flux, i.e. définir un listener capable de gérer ces trois possibilités. Un tel listener sera appelé un observer, et le flux, un observable. Ces termes ont été définis il y longtemps, car ils constituent un design pattern bien connu : l'observer.

Ils sont différents des promises, même s'ils y ressemblent, car ils gèrent tous deux des valeurs asynchrones. Mais un observer n'est pas une chose à usage unique : il continuera d'écouter jusqu'à ce qu'il reçoive un événement de terminaison. Pour le moment, les observables ne font pas partie de la spécification ECMAScript officielle, mais ils feront peut-être partie d'une version future, un effort en cours va dans ce sens.

Les observables sont très similaires à des tableaux. Un tableau est une collection de valeurs, comme un observable. Un observable ajoute juste la notion de valeur reportée dans le temps : dans un tableau, toutes les valeurs sont disponibles immédiatement, dans un observable, les valeurs viendront plus tard, par exemple dans plusieurs minutes.

La bibliothèque la plus populaire de programmation réactive dans l'écosystème JavaScript est RxJS. Et c'est celle choisie par Angular.

Concrétement qu'est-ce que cela signifie, à notre niveau dans la formation. La programmation réactive est un sujet à part entière qui prend quelques temps à appréhender et que nous n'aurons pas le temps de d'aborder. Je vais vous donner quelques exemples qui devraient vous suffire

réaliser les TP's.

```
 Voici quelques liens utiles si vous souhaitez appronfir le sujet par
vous même.
* http://reactivex.io/intro.html[ReactiveX] Site officiel
* https://github.com/ReactiveX/rxjs[RxJS]
* https://xgrommx.github.io/rx-book/index.html[Rx Book]
* http://rxmarbles.com/[rxmarbles]: Représentation graphique de chaque
fonction
```

- Un Observable possède une méthode subscribe() qui peut prendre 3 arguments :
  - une fonction pour écouter les événements.
  - une fonction pour écouter les erreurs.
  - une fonction pour écouter la terminaison

```
Observable.create([]).subscribe(
      (value) => {
        console.log('Next Value', value)
      },
      (e) => {
        console.log('Error', e)
      },
      () => {
        console.log('End');
      }
    )
```

- Un Observable possède une méthode toPromise() qui permet de transformer l'Obeservable en Promise

```
Observable.create([]).toPromise()
```

5.3. TODO Application

- Reprenez encore une fois l'application TODO
- La dernière étape, que vous avez réalisée, a été de créer un service TodoService qui ajoute et supprime les Todos. Nous allons maintenant appeler un serveur pour ajouter, supprimer, mettre à jour des Todos, mais aussi récupérer la liste des Todos.
- Je vous fournis dans le Google drive un petit serveur écrit en Node.js : todo-api. Récupérez ce dossier.
- Dans un terminal, aller dans le dossier todo-api et exécutez les commandes suivantes:

```
# Installer les dépendances
npm install

# Démarrer le serveur
npm start
```

- Dans votre navigateur: http://localhost:3000/todos vous permet de récupérer tous les Todos.
- Pour avoir un exemple de toutes les requêtes disponibles, un fichier `todos-api.postman_collection` vous ait fourni. Vous devez installer `Postman` pour importer ce fichier et tester les requêtes.
- Dans le projet Todo, créez un service `ApiService` qui aura la responsabilité de faire les appels réseaux.
- A l'initialisation de l'application, récupérez les todos sur le serveur.
- Lorsque vous ajoutez, modifiez, supprimez un todo, mettez vous données à jour sur le serveur
- Réalisez un interceptor qui positionne le header Content-Type 'application.json' sur toutes les requêtes.

Chapitre 6. Angular: Router

6.1. Concept

Une application web développée avec Angular est une SPA (Single Page Application). Cela implique que seule la première page est chargée depuis le serveur. Pourtant il est souhaitable et souhaitée que notre application réagisse en fonction d'URL saisie dans le navigateur.

Avec une application web qui n'est pas une SPA, en saisissant une URL dans le navigateur, je souhaite accéder à une page. Ceci n'est pas possible avec une application Angular, le point d'entrée étant toujours l'index.html. Pourtant le besoin est la.

Dans une application Angular, une URL représente un état (state) de l'application. Cette URL associée à un état de l'application, c'est le rôle du `RouterModule`. Plus concrétement, avec le router, nous définissons une URL (path) et le composant qui doit être affiché lorsque celle ci est appelée.

Le Router est un composant central et complexe. Je ne couvrirai pas toutes les fonctionnalités avancées du Router dans cette formation. Si vous souhaitez traiter le sujet vraiment en profondeur, Victor Savkin, core contributor du router à écrit un livre sur le sujet.

6.2. Premières routes

- Créez un nouveau projet
- Créez deux composants, un composant `Home` et un composant `Detail`, laissez pour l'instant les templates par défaut.
- Créez un fichier `app.routes.ts` à la racine du dossier app.

Sources: 65. app.routes.ts

```
import { Routes } from '@angular/router';
import { HomeComponent } from './routes/home/home.component';
import { DetailComponent } from './routes/detail/detail.component';
import { PATH_HOME, PATH_DETAIL } from './app.routes.constantes';

export const ROUTES: Routes = [                    ①
  { path: PATH_HOME, component: HomeComponent },    ②
  { path: PATH_DETAIL, component: DetailComponent } ③
];
```

① Création du tableau de routes du modules App

② Route Home. L'URL '/' affichera le composant Home

③ Route Home. L'URL '/detail' affichera le composant Detail

Une route se définit à minima avec un path (quel sera le chemin) et un composant (qui sera affiché pour ce chemin)

Sources: 66. app.routes.constantes.ts

```
export const PATH_HOME = '';
export const PATH_DETAIL = 'detail';
```

- Si vous testez l'application à ce stade, rien n'a évolué. Vous devez précisez quel est le conteneur qui contiendra le composant activé en fonction de l'état ou de l'URL.

Le conteneur est une directive fournie par le router. `<router-outlet></router-outlet>`. Sans cette directive, le router ne pourra pas afficher le composant défini par la route

- Ajoutez cette directive dans le composant racine de l'application
- Si vous testez, rien de ne s'affiche !!! En effet, pour finir la configuration vous devez déclarer le module RouterModule avec le tableau de routes dans le module AppModule

Sources: 67. app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {RouterModule} from '@angular/router';

import { AppComponent } from './app.component';
import { HomeComponent } from './routes/home/home.component';
import { DetailComponent } from './routes/detail/detail.component';
import { ROUTES } from './app.routes.hierarchic'; ①
import { DetailHierarchicComponent } from './routes/detail-hierarchic/detail-
hierarchic.component';
import { HomeHierarchicComponent } from './routes/home-hierarchic/home-
hierarchic.component';
import { FormationsComponent } from './formations/formations.component';
import { SkillsComponent } from './skills/skills.component';
import {FormationsResolvers} from './resolvers/FormationsResolvers';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    DetailComponent,
    DetailHierarchicComponent,
    HomeHierarchicComponent,
    FormationsComponent,
    SkillsComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(ROUTES)
  ],
  providers: [
    FormationsResolvers
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

① supprimer la partie `.hierarchic` pour le moment

- Dans le navigateur, l'url http://localhost:4200/ doit vous afficher le composant Home et l'url http://localhost:4200/detail le composant detail

6.3. Navigation

Nous venons de définir deux routes dans l'application qui réagissent aux URL's, mais comment naviguer entre elles dans l'application.

Dans une application web classique, nous avons deux possibilités pour naviguer vers une autre page :

- avec une balise `a` dans une page HTML. Par exemple : `<a href="ma-seconde-page.html">lien</a>`
- avec `window.location` en JavaScript

Dans une application Angular, en utilisant ces techniques, vous allez recharger complètement l'application. Vous perdrez le contexe en cours. Nous éviterons, de fait, ces approches techniques.

Le module Router d'Angular nous permet d'y remédier.

6.3.1. HTML: routerLink

Dans l'exemple ci-dessous, nous déclarons dans notre balise `a`, un attribut `routerLink` permettra au Router de calculer la bonne route.

Sources: 68. app.component.html

```
<nav class="nav">
  <span class="nav__menu">
    <a href=""
       routerLink="/">Home</a>
  </span>
  <span class="nav__menu">
    <a href=""
       routerLink="/detail">Detail</a>
  </span>
</nav>

<!--Équivalent à -->

<nav class="nav">
  <span class="nav__menu">
    <a href=""
       [routerLink]="['/']">Home</a>
  </span>
  <span class="nav__menu">
    <a href=""
       [routerLink]="['/detail']">Detail</a>
  </span>
</nav>
```

Le slash de début dans le chemin est nécessaire. S'il n'est pas inclus, RouterLink construit une URL relativement au chemin courant.

il est également possible de se servir de la directive `routerLinkActive` qui permet d'activer ou non une classe CSS si vous êtes sur cette route.

Sources: 69. app.component.html

```
<nav class="nav">
  <span class="nav__menu">
    <a href=""
       routerLink="/"
       routerLinkActive="nav__item_selected">Home</a>
  </span>
  <span class="nav__menu">
    <a href=""
       routerLink="/detail"
       routerLinkActive="nav__item_selected">Detail</a>
  </span>
</nav>
```

- Mettez en place la navigation dans le TP commencé précédemment.

6.3.2. TypeScript: navigate

Il est également possible de naviguer vers une route depuis le code TypeScript. Le RouterModule met à disposition un service `Router` qui expose une méthode navigate.

Sources: 70. app.component.ts

```
import {Component, OnInit} from '@angular/core';
import {Router} from '@angular/router';
import {PATH_DETAIL, PATH_HOME} from './app.routes.hierarchic';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'app';

  constructor(private router: Router) {}


  navigateToHome() {
    this.router.navigate([PATH_HOME]);
  }

  navigateToDetail() {
    this.router.navigate([PATH_DETAIL, this.idDetail]);
  }

  ngOnInit() {
    this.router.events.subscribe((event) => {
      console.log('EVENTS', event);
    });
  }

}
```

- Ajoutez un bouton dans la page Home qui au click vous dirige sur la page détail. Et faites l'inverse dans la page détail. Réaliser cette opération via **typescript** et non dans le template

6.3.3. Routes dynamiques

Quand nous navigons dans une application, nous avons souvent le besoin de passer des paramètres d'une page à une autre. En parallèle, ll existe aussi une bonne pratique qui consiste à faire en sorte que l'URL représente que ce nous affichons, comme par exemple :

`http://bewizyu.com/formations/2/formateur/5`

Cette URL signifie que j'affiche la page du formateur avec l'ID 5 de la formation ayant l'ID 2. Le Router nous permet donc de récupérer ces paramètres mais aussi de les transmettre.

Pour cela dans la définition du tabeau de routes, nous allons définir le chemin de la façon suivante :

`formations/:idFormation/formateur/:idFormateur`

La syntaxe `:` indique au Router que ceci sera un paramètre. Un paramètre définit dans le chemin d'une route sera toujours de type string.

Encore une fois, il est possible de le faire aussi bien coté HTML que TypeScript.

HTML

Voici ce que cela donnerai en HTML

```
    <a href="" [routerLink]="['/formations', formation.id, 'formateur', formateur.id
]">Detail</a>
```

L'URL sera construite en prenant les arguments du tableau dans l'ordre. Vous pouvez remarquer que formation et formateur sont des propriétés déclarées dans le composant TypeScript.

- Définissez une propriété idDetail dans le composant qui manage la navigation avec une valeur par défaut à 1.
- Faites évoluer votre code pour que la page détail puisse recevoir un idDetail (on ne travaille que sur le container appelant à ce stade)
- Mettez à jour votre lien pour naviguer vers la page détail

Sources: 71. app.component.html

```
<nav class="nav">
  <span class="nav__menu">
    <a href=""
       [routerLink]="['/']">Home</a>
  </span>
  <span class="nav__menu">
    <a href=""
       [routerLink]="['/detail', idDetail]">Detail</a>
  </span>
</nav>
```

TypeScript

La même exemple coté composant

```
    this.router.navigate(['/formations', formation.id, 'formateur', formateur.id]);
```

- Faites la mise à sur la navigation depuis vos boutons

Sources: 72. app.component.ts

```
import {Component, OnInit} from '@angular/core';
import {Router} from '@angular/router';
import {PATH_DETAIL, PATH_HOME} from './app.routes.hierarchic';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'app';

  constructor(private router: Router) {}

  idDetail = 1;

  navigateToHome() {
    this.router.navigate([PATH_HOME]);
  }

  navigateToDetail() {
    this.router.navigate([PATH_DETAIL, this.idDetail]);
  }

  ngOnInit() {
    this.router.events.subscribe((event) => {
      console.log('EVENTS', event);
    });
  }

}
```

Récupérer les paramètres

Voici comment récupérer dans le composant détail le paramètre:

Sources: 73. detail.component.ts

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

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

  constructor(private route: ActivatedRoute) { }   ①

  ngOnInit() {
    console.log('idDetail snapshot', this.route.snapshot.paramMap.get('idDetail')); ②

  }
}
```

① Injection du service ActivatedRoute pour récupérer les paramètres

② Récupération des paramètres vie le `snapshot.paramMap`

Que représente le snapshot ?? Comme son nom l'indique c'est une version instantanée de la route. Le routeur va recycler le composant de la route si vous appelez depuis cette route la même route avec un paramètre différent. Si nous reprenons l'exemple de la route avec les formations et le formateur. Imaginez faire un lien vers le formateur suivant. Le composant existera déja, le router va le recycler et donc comme il est déjà créé, le `ngOnInit` ne sera pas rappelé. Vous ne pourrez donc pas récupérer la nouvelle valeur du paramètre.

Pas de soucis, le Router nous permet un alternative, avec un Observable encore une fois !

Sources: 74. detail.component.ts

```
    this.route.paramMap.subscribe((params: ParamMap) => {
      console.log('isDetail', params.get('idDetail'));
    });
```

- Dans le composant detail, loguez le paramètre qui est passé par l'URL en utilisant la méthode `snapshot`
- Ajoutez dans le composant detail, un lien vers la route détail avec un idDetail valant 2

- Vérifiez que vous n’arrivez pas à loguez la valeur 2 lorsque vous cliquez sur le lien
- Mettez en place l’Observable pour récupérer les paramètres dans tous les cas de figures.

6.4. Routes hiérarchiques

Un route peut avoir des routes enfants, ce qui se nomme route hiérarchique. Ok mais pourquoi ?

- un template commun peut être défini pour plusieurs routes
- une partie du site ne peut pas être accessible pour un utilisateur. Exemple la partie administration d'un site ne sera accessible que pour un utlisateur admin
- utilisation des resolvers (nous reviendrons dessus plus loin) pour une groupe de route

Nous allons donc faire évoluer notre projet pour manipuler cette notion.

- Créez un composant Formations et un composant Connaissances
- Dans le composant Formations affichez une liste de formation
- Dans le composant Connaissances, affichez une liste de connaissances
- Définissez comme dans cet exemple ces composants en routes filles de la page detail

Sources: 75. app.routes.hierarchic.ts

```
import { Routes } from '@angular/router';
import {HomeComponent} from './routes/home/home.component';
import {DetailHierarchicComponent} from './routes/detail-hierarchic/detail-
hierarchic.component';
import {FormationsComponent} from './formations/formations.component';
import {SkillsComponent} from './skills/skills.component';
import {FormationsResolvers} from './resolvers/FormationsResolvers';

export const PATH_HOME = '';
export const PATH_DETAIL = 'detail/:idDetail';
export const PATH_DETAIL_FORMATIONS = 'formations';
export const PATH_DETAIL_SKILLS = 'connaissances';

export const ROUTES: Routes = [
  { path: PATH_HOME, component: HomeComponent },
  {
    path: PATH_DETAIL,
    component: DetailHierarchicComponent,
    children: [
      { path: '', pathMatch: 'full', redirectTo: PATH_DETAIL_FORMATIONS },
      { path: PATH_DETAIL_FORMATIONS, component: FormationsComponent },
      { path: PATH_DETAIL_SKILLS, component: SkillsComponent },
    ]
  }
];
```

- dans le template du composant detail, il faut ajouter la directive `<router-outlet></router-outlet>` pour injecter les composants

- Dans votre barre de navigation, ajoutez deux liens qui permettent de naviguer vers la page formation et connaissances

6.5. Guards

Toutes les routes d'une application ne sont pas forcément accessibles pour un utilisateur. Par exemple, il se peut qu'il faille être authentifié pour y accéder, ou bien même authentifié avec un droit d'admin.

Pour cela, le Router met à disposition un système nommé `Guards`.

Ce système bloquera l'accès à certaines routes, il faut bien sur en complément coté API Rest avoir un contrôle via une couche de sécurité dédiée.

Il existe 4 types de guards :

- `CanActivate` : lorsqu'un tel guard est appliqué à une route, il peut empêcher l'activation de la route. Il peut aussi avoir un effet de bord, comme par exemple naviguer vers une autre route. C'est ce qui permet d'afficher une page d'erreur, ou de naviguer vers la page de connexion lorsqu'un utilisateur anonyme tente d'accéder à une page qui requiert une authentification ;
- `CanActivateChild` : Ce guard peut empêcher les activations des enfants de la route sur lequel il est appliqué. Cela peut être utile, par exemple, pour empêcher l'accès à de nombreuses routes d'un seul coup, en fonction de leur URL ;
- `CanLoad` : ce guard peut être utilisé sur une route qui a un attribut loadChildren. Cet attribut permet de télécharger un module applicatif à la demande (lazy loading), contenant des routes filles.
- `CanDeactivate` : ce guard est différent des trois autres. Il est utilisé pour empêcher de quitter la route actuelle. Cela peut être utile pour, par exemple, demander une confirmation avant de quitter une page contenant un long formulaire.

Voici comment appliquer un guard CanActivate à une route. Un Guard est un service Angular qui implémente l'interface `CanActivate`. Le routeur vous dirigera sur la route si le guard retourne true.

```
{ path: 'secure', component: SecureComponent, canActivate: [LoggedInGuard] }
```

Sources: 76. logged-in-guard.guard.ts

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from
'@angular/router';
import { Observable } from 'rxjs';

@Injectable()
export class LoggedInGuardGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return true;
  }
}
```

- Créez un guard LoggedIn qui retoune false
- Ajoutez ce guard sur la route detail / connaissances
- Essayez de naviguez sur cette page
- Modifiez le return du guard à true
- Essayez de naviguez sur cette page

6.6. Resolver

Il se peut que l'on souhaite, pour des raisons fonctionnelles, récupérer des données avant de naviguer vers une page. Pourquoi ?

Si nous naviguons tout de suite vers une page, le router nous l'affiche immédiatement. Cependant il y aura surement des données à afficher et il faudra gérer alors via des loaders, par exemple, une phase d'initialisation des la page.

Que se passe-t-il si un appel réseau échoue ? C'est un cas métier qu'il faut bien gérer.

Angular nous propose une possibilité technique autre, qui consiste à récupérer les données nécessaires avant d'afficher une page. Cela se nomme un `resolver`.

- Définissez un resolver `FormationsResolvers` qui retourne la liste de formation pour la page de formations

Sources: 77. FormationsResolvers.ts

```
import {ActivatedRouteSnapshot, Resolve, RouterStateSnapshot} from '@angular/router';
import {Observable} from 'rxjs';
import {Injectable} from '@angular/core';

@Injectable()
export class FormationsResolvers implements Resolve<any> {

  constructor() {
  }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<any>
| Promise<any> | any {
    return [{
      title: 'Module Angular',
      decription: '',
    }, {
      title: 'Module JavaScript',
      decription: '',
    }, {
      title: 'Module TypeScript',
      decription: '',
    }];
  }

}
```

Dans mon exemple la récupération de la liste est synchrone mais retournez la avec une Promise, qui renvoi les données dans 5 secondes.

- Ajoutez la configuration du resolver dans la route DetailFormations

```
resolve:{
  formations: FormationsResolvers
}
```

- récupérez les données dans le composant pour les afficher

```
this.route.data.subscribe(data => console.log('Formations', data['formations']));
```

6.7. Events

Le Router émet toute une série d'événements pendant la navigation. Il est possible d'écouter ces évènements et de réagir dessus.

C'est l'objet `Router` qui émet via l'observable events

```
    this.router.events.subscribe((event) => {
          console.log('EVENTS', event);
        })
```

Voici les différents types d'événements principaux :

- `NavigationStart` : émis lorsqu'une navigation est demandée.
- `NavigationEnd` : émis lorsque la navigation s'achève avec succès.
- `NavigationError` : émis lorsque la navigation échoue.
- `NavigationCancel` : émis lorsqu'une navigation est annulée.
    - Loguez les événements du routeur

6.8. TP

En soignant le style (les éléments doivent être bien positionnés).

- Créez une nouvelle application
- Sur la HomePage, ajoutez un champ de saisie afin de récupérer le login de l'utilisateur Github avec un bouton valider
- Si l'appel est en erreur resté sur la même page et affichez un message d'erreur
- Sinon naviguez vers une page de détail de l'utilisateur
- Affichez les informations suivantes (login, type, compagny, bio, location et l'avatar)
- Cette page de détail comporte 2 tab en dessous de la description
  - repos: affiche les repo d'un user
  - followers: affiche les follower d'un user

URL pour récupérer les informations nécessaire d'un user Sample

Chapitre 7. Angular: Formulaires

7.1. Généralités

L'ensemble des directives, composants et services fournis par Angular est porté par la dépendance `@angular/forms`. Cette dépendance doit être déclarée en dépendance de l'application. Vous devez importer le module `FormsModule` dans le module de l'application.

Une fois le module ajouté, Angular permet de créer des formulaires de deux manières:

- template first: privilégié pour réaliser des formulaires simples, déclaration dans le template
- code first: priviligié pour des formulaires plus complexes. Permet de définir des validateurs personnalisés, des ajouts de champs dynamiquement, ..

Quelque soit l'approche que vous utiliserez, Angular crée des objets qui modèlisent, représentent le formulaire, notamment des composants de type `FormControl` et `FormGroup`. Avec l'approche par le template, les directives permettent de créer ces objets et coté code, nous les créons dans le composant et Angular les 'bind' sur les champs du template.

7.1.1. FormControl

Un champ de formulaire de type `input` par exemple sera représenté par un FormControl. C'est le plus petit élément de la représentation. Via cet objet vous pourrez accéder, à sa valeur, aux erreurs de validations du champ, et à toutes les informations nécessaires. Le champ est-il valide, invalide, a-t-il déjà modifié la valeur, pris le focus sur le champ, ...

Voici la liste complète des propriétés d'un objet FormControl:

- `value` : la valeur du champ.
- `valueChanges` : un Observable qui émet à chaque modification du champ.
- `valid` : le champ est-il valide ?
- `invalid` : le champ est-il invalide ?
- `errors` : Récupération des erreurs du champ
- `dirty` : = false jusqu'à ce que l'utilisateur modifie la valeur du champ.
- `pristine` : = true jusqu'à ce que l'utilisateur modifie la valeur du champ
- `touched` : = false tant que l'utilisateur n'a pas pris le focus sur le champ.
- `untouched` : = true tant que l'utilisateur n'a pas pris le focus sur le champ.
- `hasError`: fonction qui permet de connaitre si le champ a une erreur donnée. Prend le nom de l'erreur en paramètre

7.1.2. FormGroup

Un `FormGroup` est un objet qui permet de grouper les contrôles de type FormControl. Un formulaire est un FormGroup par exemple. Il est possible comme avec le FormControl de savoir si le groupe est valide, possède une erreur, ... A peu de chose près, un FormGroup a les mêmes propriétés qu'un FormControl:

- `valid` : le champ est-il valide ?
- `invalid` : le champ est-il invalide ?
- `errors` : Récupération des erreurs du champ
- `dirty` : = false jusqu'à ce que l'utilisateur modifie la valeur du champ.
- `pristine` : = true jusqu'à ce que l'utilisateur modifie la valeur du champ
- `touched` : = false tant que l'utilisateur n'a pas pris le focus sur le champ.
- `untouched` : = true tant que l'utilisateur n'a pas pris le focus sur le champ.
- `hasError`: fonction qui permet de connaitre si le champ a une erreur donnée. Prend le nom de l'erreur en paramètre
- `get`: fonction qui permet de récupérer un FormControl par son nom. Prend le nom du FormControl en paramètre.

Nous allons manipuler le même formulaire (champ email et password) avec les deux techniques, puis nous les combinerons.

7.2. Template first

- Créez un nouveau projet
- Ajoutez l'import du `FormsModule` dans le module applicatif

Sources: 78. app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {RouterModule} from "@angular/router";

import { AppComponent } from './app.component';
import {ROUTES} from './app.routes';
import { TemplateComponent } from './routes/template/template.component';
import { TypescriptComponent } from './routes/typescript/typescript.component';
import { ValidatorsComponent } from './routes/validators/validators.component';
import { StylesComponent } from './routes/styles/styles.component';
import {FormsModule} from "@angular/forms";
import { TwoWayBindingComponent } from './routes/two-way-binding/two-way-
binding.component';

@NgModule({
  declarations: [
    AppComponent,
    TemplateComponent,
    TypescriptComponent,
    ValidatorsComponent,
    StylesComponent,
    TwoWayBindingComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(ROUTES),
    FormsModule ,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

7.2.1. ngModel

- Créez le formulaire suivant dans un composant que vous importerez dans le component de l'application

Sources: 79. template.component.html

```
<form (ngSubmit)="handleSubmit()"> ①
  <div>
    <label>Email: </label>
    <input name="email"
           ngModel> ②
  </div>
  <div>
    <label>Password: </label>
    <input type="password"
           name="password"
           ngModel>
  </div>
  <button type="submit">Login</button> ③
</form>
```

① Ajout de la directive ngSubmit qui déclenchera la fonction handleSubmit qui est définit dans le composant

② Ajout de la directive ngModel qui crée le composant FormControl. Vous devez définir un attribut `name` au champ input pour récupérer sa valeur. Angular s'en sert aussi pour créer le FormGroup.

③ Bouton de soumission du formulaire.

La balise `<form>` crée l'objet FormGroup qui représente le formulaire

7.2.2. Récupération des valeurs des champs

- Faites évoluer votre formulaire comme suit:

Sources: 80. template.component.html

```
<form (ngSubmit)="handleSubmit(loginForm.value)"
      #loginForm="ngForm"> ①
  <div>
    <label>Email: </label>
    <input name="email"
           ngModel>
  </div>
  <div>
    <label>Password: </label>
    <input type="password"
           name="password"
           ngModel>
  </div>
  <button type="submit">Login</button>
</form>
```

① Création d'une variable locale loginForm qui prend la valeur de ngForm. Vous pouvez ensuite transmettre sa valeur en argument de la fonction handleSubmit

- Loguez la valeur dans le composant

Sources: 81. template.component.ts

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

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

  constructor() {
  }

  ngOnInit() {
  }

  handleSubmit(value) {
    console.log('Form value', value);
  }

}
```

- Voici ce que vous allez recevoir dans la console

```
    Form value {email: "nartawak", password: "test"}
```

- Vous pouvez constatez que `loginForm.value` est un objet qui possède en propriété les noms de champs inputs du formulaire.

7.2.3. Binding bi-directionnel

Actuellement tout le formulaire est décrit de manière déclarative dans le template. En aucun cas nous ne pouvons accéder à la valeur du champ dans le composant. La seule façon de récupérer cette valeur consiste à la transmettre comme nous l'avons vu précédemment.

Nous allons faire évoluer encore une fois le formulaire.

- Créez une classe User. Afin de loguer tout ce qui se passe quand la valeur de ces propriétés change, nous allons déclarer ses propriétés en privée avec des getters / setters.

Sources: 82. user.model.ts

```
export class User {

  private _email:string;
  private _password:string;

  constructor(_email?:string, _password?:string){
    this._email = _email;
    this._password= _password;
  }

  get email(): string {
    return this._email;
  }

  set email(value: string) {
    console.log('SET EMAIL', value);
    this._email = value;
  }
  get password(): string {
    return this._password;
  }

  set password(value: string) {
    console.log('SET PASSWORD', value);
    this._password = value;
  }


}
```

- Ajoutez une propriété user de type User dans le composant

Sources: 83. two-way-binding.component.ts

```
import { Component, OnInit } from '@angular/core';
import {User} from "../../models/user.model";

@Component({
  selector: 'app-two-way-binding',
  templateUrl: './two-way-binding.component.html',
  styleUrls: ['./two-way-binding.component.css']
})
export class TwoWayBindingComponent implements OnInit {

  user: User = new User('Nartawak', 'test');

  constructor() { }

  ngOnInit() {
  }

  handleSubmit() {
    console.log('SUBMIT', this.user);
  }

}
```

- Faites évoluer le template comme suit:

Sources: 84. two-way-binding.component.html

```
<form (ngSubmit)="handleSubmit()">
  <div>
    <label>Email: </label>
    <input name="email"
           [ngModel]="user.email"
           (ngModelChange)="user.email =$event">
  </div>
  <div>
    <label>Password: </label>
    <input type="password"
           name="password"
           [ngModel]="user.password"
           (ngModelChange)="user.password =$event">
  </div>
  <button type="submit">Login</button>
</form>
```

- Vous pouvez remarquer que lors de l'initialisation du formulaire, les valeurs du user sont 'bind' dans le template. Si vous modifiez la valeur de ces champs de saisies, à chaque changement de

valeur, le model est mis à jour.

En utilisant les syntaxes de binding de propriété et de binding d'événement, nous pouvons donc mettre à jour le modèle que ce soit dans le template et dans le composant. La directive `ngModelChange` émet un événement à chaque changement de valeur.

Il existe une syntaxe plus concise pour éviter cette déclaration un peu verbeuse. C'est une aggrégation des deux syntaxes de binding ⇒ `[()]`. Cette syntaxe est ce que l'on nomme le **binding bi-directionnel**.

L'exemple suivant donne le même résulat que la version que vous venez d'écrire. Nous le privilégierons donc ;)

Sources: 85. two-way-binding.component.html

```
<form (ngSubmit)="handleSubmit()">
  <div>
    <label>Email: </label>
    <input name="email"
           [(ngModel)]="user.email">
  </div>
  <div>
    <label>Password: </label>
    <input type="password"
           name="password"
           [(ngModel)]="user.password">
  </div>
  <button type="submit">Login</button>
</form>
```

7.3. TypeScript first

- Ajoutez l'import du `ReactiveFormsModule` dans le module applicatif. Ce module nous permet d'utiliser tout un nouveau jeu de directives. Tout comme le `FormsModule`, il est disponible via la dépendance `@angular/forms`.

Sources: 86. app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {RouterModule} from "@angular/router";

import { AppComponent } from './app.component';
import {ROUTES} from './app.routes';
import { TemplateComponent } from './routes/template/template.component';
import { TypescriptComponent } from './routes/typescript/typescript.component';
import { ValidatorsComponent } from './routes/validators/validators.component';
import { StylesComponent } from './routes/styles/styles.component';
import {FormsModule, ReactiveFormsModule} from "@angular/forms";
import { TwoWayBindingComponent } from './routes/two-way-binding/two-way-
binding.component';

@NgModule({
  declarations: [
    AppComponent,
    TemplateComponent,
    TypescriptComponent,
    ValidatorsComponent,
    StylesComponent,
    TwoWayBindingComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(ROUTES),
    FormsModule ,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

- Créez un nouveau composant dans l'application
- Mettez à jour votre composant en prenant exemple sur le bout de code suivant:

Sources: 87. typescript.component.ts

```
import { Component, OnInit } from '@angular/core';
import {FormBuilder, FormControl, FormGroup} from "@angular/forms";

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

  emailCtrl: FormControl;
  passwordCtrl: FormControl;
  userForm: FormGroup;

  constructor(fb: FormBuilder) {      ①

    // Création des contrôles
    this.emailCtrl = fb.control(''); ②
    this.passwordCtrl = fb.control('');

    // Création du groupe (aka le formulaire)
    this.userForm = fb.group({    ③
      email: this.emailCtrl,
      password: this.passwordCtrl
    });
  }

  handleClear() {
    this.emailCtrl.setValue('');    ④
    this.passwordCtrl.setValue('');
  }

  handleSubmit() {
    console.log(this.userForm.value);     ⑤
  }

  ngOnInit() {
  }

}
```

① Injection du service utilitaire FormBuilder

② Création d'un contrôle via le FormBuilder

③ Création du group via le FormBuilder

④ Il est possible de mettre à jour la valeur d'un contrôle via la fonction `setValue`

⑤ Récupération des valeurs saisies dans le formulaire

- Créez Le template comme suit:

Sources: 88. typescript.component.html

```
<h4>Formulaire: typescript</h4>

<form (ngSubmit)="handleSubmit()" [formGroup]="userForm">
  <div>
    <label>Email: </label>
    <input formControlName="email">
  </div>
  <div>
    <label>Password: </label>
    <input type="password"
           formControlName="password">
  </div>
  <button type="submit">Login</button>
  <button type="reset" (click)="handleClear()">Clear</button>
</form>
```

Les directives `formGroup` et `formControlName` permettent de faire le lien entre les propriétés du composant, le formulaire et les champs de saisies.

7.4. Valider les champs

La validation des champs s'effecue via des Validators. Un Validator permet de valider qu'un champ est conforme à ce que nous souhaitons obtenir comme informations dans un champ donné. Par exemple, nous voulons que le champ `email` soit bien un email valide et qu'il soit absolument saisi (required).

7.4.1. Code first

Angular met à disposition quelques validators qu'il est possible de déclarer sur un FormControl:

- `Validators.required`
- `Validators.minLength(n)`
- `Validators.maxLength(n)`
- `Validators.email()`
- `Validators.pattern(p)`
- `Validators.min(p)`
- `Validators.max(p)`
- Nous allons continuer le formulaire précédent, sur le champ email, ajoutez un validator email et required

Sources: 89. validators.component.ts

```
import { Component, OnInit } from '@angular/core';
import {FormBuilder, FormControl, FormGroup, Validators} from "@angular/forms";
import {isPrefixNartawakValidator} from "../../validators/password.validator";

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

  emailCtrl: FormControl;
  passwordCtrl: FormControl;
  userForm: FormGroup;

  constructor(fb: FormBuilder) {

    // Création des contrôles
    this.emailCtrl = fb.control('', [Validators.email, Validators.required]); ①
    this.passwordCtrl = fb.control('');

    // Création du groupe (aka le formulaire)
    this.userForm = fb.group({
      email: this.emailCtrl,
      password: this.passwordCtrl
    });
  }

  handleClear() {
    this.emailCtrl.setValue('');
    this.passwordCtrl.setValue('');
  }

  handleSubmit() {
    console.log(this.userForm.value);
  }

  ngOnInit() {
  }

}
```

① Ajout des Validators email et required

- Affichez l'erreur dans le template si le champ n'est pas un pas un email valide et si il n'est pas renseigné. Il est une bonne pratique, qui consiste à interdire à l'utilisateur de valider le formulaire si celui-ci n'est pas valide en désactivant le bouton. Implémentez ce

comportement

Sources: 90. validators.component.html

```
<form (ngSubmit)="handleSubmit()"
      [formGroup]="userForm">
  <div>
    <label>Email: </label>
    <input formControlName="email">
    <div *ngIf="userForm.get('email').hasError('email')">
      Le champ email n'est pas valide
    </div>
    <div *ngIf="userForm.get('email').hasError('required')">
      Le champ email est requis
    </div>
  </div>
  <div>
    <label>Password: </label>
    <input type="password"
           formControlName="password">
  </div>
  <button type="submit"
          [disabled]="userForm.invalid">Login
  </button>
  <button type="reset"
          (click)="handleClear()">Clear
  </button>
</form>
```

- En testant, vous vous rendrez compte que les messages d'erreurs sont affichés dès l'initialisation du formulaire. Il faut adapter la condition d'affichage avec la propriété `dirty` du contrôle.

Sources: 91. validators.component.html

```
<form (ngSubmit)="handleSubmit()"
      [formGroup]="userForm">
  <div>
    <label>Email: </label>
    <input formControlName="email">

    <div *ngIf="userForm.get('email').dirty &&
userForm.get('email').hasError('email')">
      Le champ email n'est pas valide
    </div>
    <div *ngIf="userForm.get('email').dirty &&
userForm.get('email').hasError('required')">
      Le champ email est requis
    </div>

    <!--équivalent à (mais plus court à écrire) -->

    <div *ngIf="emailCtrl.dirty && emailCtrl.hasError('email')">
      Le champ email n'est pas valide
    </div>
    <div *ngIf="emailCtrl.dirty && emailCtrl.hasError('required')">
      Le champ email est requis
    </div>
  </div>
  <div>
    <label>Password: </label>
    <input type="password"
           formControlName="password">
  </div>
  <button type="submit"
          [disabled]="userForm.invalid">Login
  </button>
  <button type="reset"
          (click)="handleClear()">Clear
  </button>
</form>
```

7.4.2. Template first

Il est également possible de faire le même traitement avec une approche template first. Voici le même exemple que précédemment:

Sources: 92. validators.component.html

```
<form (ngSubmit)="handleSubmit(userFormTemplate.value)"
      #userFormTemplate="ngForm">
  <div>
    <label>Email: </label>
    <input name="emailTemplate"
           ngModel
           required
           email
           #emailTemplate="ngModel">
    <div *ngIf="emailTemplate.dirty && emailTemplate.hasError('email')">
      Le champ email n'est pas valide
    </div>
    <div *ngIf="emailTemplate.dirty && emailTemplate.hasError('required')">
      Le champ email est requis
    </div>
    <div>
      <label>Password: </label>
      <input type="password"
             name="passwordTemplate"
             ngModel
             required
             #passwordTemplate="ngModel">
    </div>
    <button type="submit"
            [disabled]="userFormTemplate.invalid">Login
    </button>
    <button type="reset"
            (click)="handleClear()">Clear
    </button>
  </div>

</form>
```

7.4.3. Création d'un Validator personnalisé

- Il est bien sûr possible de créer ses propres validators. Voici comment en créer un :

Sources: 93. password.validator.ts

```
import {FormControl} from "@angular/forms";
import _ from 'lodash';

export const NARTAWAK_PREFIX:string = 'nk'
export const PREFIX_SEPARATOR:string = '-'

/**
 * Le valeur du champ commennce-t-elle par nk-
 * @param {FormControl} control
 * @returns {{isPrefixNartawak: boolean}}
 */
export function isPrefixNartawakValidator (control: FormControl)  {
  const prefix = _.split(control.value, PREFIX_SEPARATOR, 1);
  return _.isArray(prefix)   && NARTAWAK_PREFIX === prefix[0] ? null : {
isPrefixNartawak: true };
};
```

- Ajoutez le ensuite dans la liste des validators du champ password par exemple:

Sources: 94. validators.component.ts

```
import { Component, OnInit } from '@angular/core';
import {FormBuilder, FormControl, FormGroup, Validators} from "@angular/forms";
import {isPrefixNartawakValidator} from "../../validators/password.validator";

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

  emailCtrl: FormControl;
  passwordCtrl: FormControl;
  userForm: FormGroup;

  constructor(fb: FormBuilder) {

    // Création des contrôles
    this.emailCtrl = fb.control('', [Validators.email, Validators.required]); ①
    this.passwordCtrl = fb.control('', [isPrefixNartawakValidator]);

    // Création du groupe (aka le formulaire)
    this.userForm = fb.group({
      email: this.emailCtrl,
      password: this.passwordCtrl
    });
  }

  handleClear() {
    this.emailCtrl.setValue('');
    this.passwordCtrl.setValue('');
  }

  handleSubmit() {
    console.log(this.userForm.value);
  }

  ngOnInit() {
  }

}
```

- Intégrez les messages d'erreur dans le template lorsque le champ est invalide

Sources: 95. validators.component.html

```
<form (ngSubmit)="handleSubmit()"
      [formGroup]="userForm">
  <div>
    <label>Email: </label>
    <input formControlName="email">

    <div *ngIf="userForm.get('email').dirty &&
userForm.get('email').hasError('email')">
      Le champ email n'est pas valide
    </div>
    <div *ngIf="userForm.get('email').dirty &&
userForm.get('email').hasError('required')">
      Le champ email est requis
    </div>
  </div>
  <div>
    <label>Password: </label>
    <input type="text"
           formControlName="password">
    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('isPrefixNartawak')">
      Le champ password ne commence pas par le préfix nk-
    </div>
  </div>
  <button type="submit"
          [disabled]="userForm.invalid">Login
  </button>
  <button type="reset"
          (click)="handleClear()">Clear
  </button>
</form>
```

7.5. Styles

A ce point, vous savez créer des formulaires, valider les champs et afficher les erreurs de validation. Il manque la dernière étape, le style. L'exemple le plus commun est de mettre les contours en rouge d'un champ lorsqu'il est invalide. Encore une fois, Angular apporte une solution.

Pour rappel, le FormControl met à disposition une série de propriétés qui nous permet de connaitre le statut d'un champ de saisie (`valid`, `invalid`, `dirty`, `pristine`, `touched`, `untouched`).

A chacune de ses propriétés corespond une classe CSS.

- `valid` => `ng-valid`
- `invalid` => `ng-invalid`
- `dirty` => `ng-dirty`
- `pristine` => `ng-pristine`
- `touched` => `ng-touched`
- `untouched` => `ng-untouched`

Il est nous donc possible de définir un style en fonction de l'état d'un FormControl. Par exemple si un des validateurs d'un champ détecte une erreur, la classe ng-invalid sera positionnée par Angular sur le champ en question.

Voici un exemple qui affiche un contour rouge sur un champ `email` lorsque celui-ci est en erreur, et que l'utilisateur a effectué une modification dans le champ.

Sources: 96. styles.component.html

```
<h4>Formulaire: Style</h4>

<form (ngSubmit)="handleSubmit()"
      [formGroup]="userForm">
  <div>
    <label>Email: </label>
    <input [formControl]="emailCtrl"
           [(ngModel)]="user.email"
            class="input-angular">
  </div>
  <div>
    <label>Password: </label>
    <input type="text"
           [formControl]="passwordCtrl"
           [(ngModel)]="user.password">
  </div>
  <button type="submit"
          [disabled]="userForm.invalid">Login
  </button>
  <button type="reset"
          (click)="handleClear()">Clear
  </button>
</form>
```

Sources: 97. styles.component.ts

```
import { Component, OnInit } from '@angular/core';
import {FormBuilder, FormControl, FormGroup, Validators} from "@angular/forms";
import {User} from "../../models/user.model";
import {isPrefixNartawakValidator} from "../../validators/password.validator";

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

  emailCtrl: FormControl;
  passwordCtrl: FormControl;
  userForm: FormGroup;

  user: User = new User('Nartawak', 'test');

  constructor(fb: FormBuilder) {

    // Création des contrôles
    this.emailCtrl = fb.control('', [Validators.email, Validators.required]);
    this.passwordCtrl = fb.control('', [isPrefixNartawakValidator]);

    this.userForm = fb.group({
      email: this.emailCtrl,
      password: this.passwordCtrl
    });

  }

  handleClear() {
    this.user.email = '';
    this.user.password = '';
  }

  handleSubmit() {
    console.log(this.user);
  }
  ngOnInit() {
  }

}
```

Sources: 98. styles.component.css

```
.input-angular {
  background-color: rgba(205, 208, 214, 0.56);
  color: #3e3c3c;
}

.input-angular.ng-invalid.ng-dirty {
  border: 2px red solid;
  border-radius: 2px;
}
```

- Reprenez le TP précédent.
- Ajoutez un contour vert sur le champ password lorsqu'il est valide et rouge lorsqu'il est invalide et modifié par l'utilisateur

7.6. TP

- Créez un nouveau projet
- Réalisez un formulaire qui permet de saisir les informations suivantes:
  - Nom => Validation: requis, commence par une majuscule
  - Prénom => Validation: requis
  - Téléphone => Validation: numéro de téléphone valide (10 chiffres)
  - Email => Validation: requis, email
  - Connaissances => composant `select`, Validation: Requis (les connaissances possibles sont JavaScript, TypeScript, Angular)
- Le bouton de validation est actif si le formulaire est valide
- Le bouton reset permet de réinitialiser tous les champs
- Les messages d'erreurs de validation sont affichés en dessous de chaque champ de saisie.
- Les champs en erreur seront avec un background rouge et le texte en blanc.

Chapitre 8. Angular: Tests

Tester doit être la première action enclenchée lors de l'initialisation d'un développement de feature, composant, fonction ... Le test sert de référence de conception de votre code, il précisera ce qui est attendu en terme d'interface ce qui fait que commencer par le test vous apporte 2 grands avantages :

1. Vous aurez les idées claires sur l'objectif à atteindre puisque décrit algorithmiquement dans vos tests
2. Vous aurez la possibilité de vérifier en permanence si l'objectif est atteint et si les contraintes sont respectées. Et ceci de manière **automatique**

Bien sûr cette vision des tests leur confère une bien plus grande importance que *J'écris mes tests à la fin pour cocher la case "Tests faits" dans ma TODO liste*. Ils sont donc à réaliser avec beaucoup de soin car **ils sont UTILES**

C'est cette approche que l'on nomme **TDD** : **T**est **D**riven **D**evelopment.

Vous allez être en mesure de réaliser 2 types de tests avec Angular :

- Les tests Unitaires
- Les tests End-to-End

8.1. Tests Unitaires

Les tests unitaires peuvent s'écrire de 2 façons :

1. Une approche traditionnelle qui consiste à appliquer des entrées définies et vérifier les sorties naturellement attendues. C'est une vérification par état assez bien adaptée à des algorithmes très cartésiens, mathématiques, simples comme par exemple les fonctions d'une calculatrice.

Dans ce cas de tests, il faut bien penser à tester en priorité les cas non-passants, les tests au limites (dépassement de mémoire, nombre très grands ou très petits, les dates, etc.)

1. Une approche beaucoup plus 'fonctionnelle' qui fait partie d'une méthode **BDD** : **B**ehavior **D**riven **D**evelopment et qui consiste à ne plus tester les états (donc les sorties) mais les interactions d'un algorithme testé avec le monde extérieur et également et surtout, décrire le contexte fonctionnel du test : quel est le cas d'usage ? Quel est l'état du système en entrée ? Quel est l'état du système en sortie ?

Par exemple si je veux tester un service Angular qui appelle une API Web, je peux vérifier que l'API est bien appelée avec les bons paramètres en fonction des entrées du service. Je ne vais pas tester le résultat de l'API externe (la liste retournée par exemple), mais vérifier que l'appel à l'API a bien été effectué. De plus celà à l'avantage de ne plus faire appel à une API externe dont la disponibilité

pourrait influer sur l'exécution du test unitaire mais de pouvoir bouchonner très simplement l'API.

Ces 2 approches sont complémentaires et on ne peut pas se passer de l'une ou l'autre !

Par évolution, les frameworks ou librairies de test proposent tous une solution élégante de faire des tests mélageant les 2 approches :

- On décrit les cas d'usages fonctionnels testés pour être explicite lors de la lecture des rapports de test, on sait ainsi quelle fonctionnalité est testée
- On décrit l'état fonctionnel du système, abstraction de son état technique interne
- On effectue des tests à la fois sur les interactions et sur les états de sortie. Pour se faire nous avons à dispositions : des mocks, des stubs, des spys ...

C'est l'ensemble de toutes ces pratiques qui sont inclues dans la méthode **BDD**

8.1.1. L'outillage de test unitaires d'Angular

Angular intègre directement sans aucune manipulation tous les outils nécessaires au test :

- Jasmine
- Karma

Jasmine

Jasmine est la librairie qui vous fournit la syntaxe d'écriture des tests et les APIs de vérifications (`assert`, `expect`...)

Karma

Karma est l'outil (outil de terminal javascript) qui vous permet de lancer automatiquement les tests unitaires de votre projet. Il est intégré et exécutable via la CLI d'Angular

8.1.2. Premier test

Reprenons notre service `FormationService` dans sa première version : appel d'une API `FormationApi` qui simule un stockage en mémoire d'une liste de formation.

Sources: 99. formation.service.ts

```
import { Injectable } from '@angular/core';
import Formation from '../model/Formation';
import FormationApi from './formation.api';

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

  constructor(private formationApi: FormationApi) { }

  addFormation(formation: Formation) {
    this.formationApi.addFormation(formation);
  }

  getFormations() {
    return this.formationApi.formations;
  }



}
```

Un test Jasmine de ce service et de la méthode `addFormation` pourrait être :

Sources: 100. formation.service.spec.ts

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

import { FormationService } from './formation.service';
import Formation from '../model/Formation';
import FormationApi from './formation.api';

describe('FormationServiceService', () => {



  beforeEach(() => {
    const apiFormation = new FormationApi();
    spyOn(apiFormation, 'addFormation').and.callThrough();  ④

    TestBed.configureTestingModule({
      providers: [
        { provide: FormationApi, useValue: apiFormation }, ⑤
        FormationService
      ]
    });
  });

  it('should be created', inject([FormationService], (service: FormationService) => {
    expect(service).toBeTruthy();
  }));

  it('should add a new formation with empty list',
    inject([FormationService, FormationApi], (service: FormationService, api:
FormationApi) => { ①

    const formation: Formation = new Formation('test', 'description', 1000);
    service.addFormation(formation); ②
    expect(service.getFormations()).toContain(formation);  ③
    expect(api.addFormation).toHaveBeenCalledWith(formation); ⑥

  }));
});
```

① On injecte le service `FormationService` dans le cas de test

② C'est l'appel à la fonction que l'on veut tester : `addFormation` : cette fonction n'a pas de retour, son résultat sera donc observé par la suite

③ On est typiquement sur un test de résultat où l'on observe le résultat de l'action `addFormation` sur l'état interne de l'API (via la méthode `getFormations`)

④ On passe maintenant à la méthode de test par interaction : on créer une instance de `FormationApi` et on place un *espion* sur la méthode `addFormation`. On précise également que l'on

souhaite malgré tout un appel à l'implémentation concrète de `addFormation` sur une `FormationApi`

⑤ Cette instance `FormationApi` est utilisée dans les `providers` et sera injectée dans le service

⑥ On teste l'interaction produite sur `api` lors de l'appel de `service.addFormation`

TP

1. Créer l'interface de la méthode `deleteFormation` dans le service `FormationService` avec une implémentation sur `FormationApi`
2. Ecrire les tests de `deleteFormation` avec plusieurs cas :
   a. La liste initiale est vide (initialisée mais vide)
   b. La liste est remplie mais ne contient pas l'objet `formation` fournie en paramètre
   c. La liste est remplie et contient l'objet `formation`
3. Ecrire l'implémentation de la méthode `deleteFormation` pour faire passer les tests
4. Créer l'interface de la méthode `updateFormation` dans le service `FormationService` avec une implémentation sur `FormationApi`. La méthode prend en paramètre 3 paramètres :
   - `oldFormation` : la formation à mettre à jour
   - `field` : le champ à mettre à jour
   - `value` : la valeur à placer dans le champ
5. Imaginez tous les cas de tests qui sont nécessaires pour valider le fonctionnement correct de la méthode `updateFormation` et les écrire
6. Implémenter la méthode `updateFormation` au niveau du service `FormationService` (et de l'api)

1. Il est possible de désactiver des tests unitaires en préfixant `describe` et `it` respectivement par `xdescribe` ou `xit`\`
2. Il est possible de n'activer que des tests unitaires particuliers en préfixant `describe` et `it` respectivement par `fdescribe` et `fit`

8.2. Tests End-to-End

Les tests end-2-end sont des tests qui vont combiner l'utilisation des composants, resolvers, services … Ils font simuler ce qu'un utilisateur final pourrait effectuer sur votre application.

Ils sont donc :

- Plus globaux et abstraits que les tests unitaires
- Couvrent simulatéement plusieurs éléments de votre application
- Sont plus lents à l'exécution qu'un test unitaire
- Sont moins utiles pour tester les cas aux limites

On utilise le même framework de test unitaire pour écrire les tests **e2e** : **Jasmine** mais on utilise une API complémentaire qui va nous permettre d'avoir accès à des éléments comme un navigateur virtuel : Protractor

Imaginons que nous souhaitons tester le message d'accueil de notre application.

1. Nous commençons par créer un nouveau fichier `app.e2e-spec.ts` dans le répertoire `e2e/src` de notre application
2. Ce fichier va contenir les tests **Jasmine** :

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

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

  beforeEach(() => {
    page = new AppPage(); ①
  });

  it('should display welcome message', () => {
    page.navigateTo(); ②
    expect(page.getParagraphText()).toEqual('Welcome to app!'); ③
  });
});
```

① Nous créeons un nouvel objet qui va encapsuler l'utilisation de **protractor** et modéliser notre page "Web" d'accueil

② Dans notre test, nous simulons l'action de l'utilisateur lorsqu'il navigue vers notre page d'accueil, c'est-à-dire lorsqu'il entre l'url de notre application dans la barre d'adresse de son navigateur

③ Un test **Jasmine** classique pour vérifier quel est le contenu d'un paragraphe spécifique de notre page d'accueil

3. Le fichier importé `e2e/sr/app.po.ts` utilise **protractor** pour donner des accès au DOM chargé par un navigateur :

```
import { browser, by, element } from 'protractor'; ①

export class AppPage {
  navigateTo() {
    return browser.get('/'); ②
  }

  getParagraphText() {
    return element(by.css('app-root h1')).getText(); ③
  }
}
```

① Import des différents éléments de l'API **protractor**

② Utilisation d'un objet `browser` puis navigation vers l'accueil de notre application : path `/`

③ Récupération de l'élément HTML qui nous intéresse (le paragraphe) grâce à l'utilisation d'un selector CSS

La commande pour lancer spécifiquement les tests e2e est :

```
$ ng e2e
```

Comme vous pourrez l'observer, les tests nécessitent l'ouverture d'un navigateur (Chrome par défaut s'il est installé) et utilise **Selenium** pour effectuer le lien entre les tests et les drivers du navigateur. Il est ainsi possible de lancer les tests simultanéement sur plusieurs navigateurs.

TP

1. Ecrire des tests e2e pour vérifier que le CRUD des formations est opérationnel d'un point de vue interface utilisateur Web

Chapitre 9. Annexes

9.1. Typescript samples

```
// ********************************* //
// ************ Types ************** //
// ********************************* //

// ***** Types primitifs ***** //

let typeNumber:number;
let typeString:string = 'Formation Angular';

// ***** Types complexes ***** //

let typeObject: object = {
    type: 'formation',
    'language': 'javscript'
};

class Formation {
    type:any
}

let typeClass:Formation = new Formation();

let formations:Array<Formation> = [new Formation()];

let typeDynamicOrUnknown:any = 12;
typeDynamicOrUnknown = 'Formation Angular';

let typeStringOrNumber:string|number = 'Formation';
typeStringOrNumber = 10;

// ***** Types Enum ***** //

enum TypeFormation {TypeScript, Angular, Cordova};
typeClass.type = TypeFormation.Cordova;

// ***** Types retour de fonction ***** //

function getFormation ():Array<Formation> {
    return formations;
}

function doSomething(): void {
    // Code ...
```

```
}

// ***** Types interfaces ***** //

// Exemple interface => forme de l'objet
function learnFormation (person: { skills : Array<Formation>}, formation: Formation
):void {}

// Exemple interface TypeScript

interface hasSkills {
    skills:Array<Formation>;
}

function learnFormationInterface (person: hasSkills, formation: Formation):void {}

interface canDrink {
    drink():void;
}

function serveBeer (person:canDrink):void {
    person.drink();
}

const person = {
    drink: () => {
    }
};

serveBeer(person);

interface ExtendInterface extends canDrink, hasSkills {
    speak():void;
}

// ***** Types Class ***** //

class Person implements ExtendInterface {
    drink() {}
    skills:Array<Formation>;                    // uniquement en TS impossible en ES6
    speak(){}
}

// Propriété privé
// Par defaut, toutes les proproiétés sont publiques. TS nous permet rendre privé une
propriété ou une function.

class Human {
```

```
    constructor(public name :string, private religion: string){

    }

    // Équivalent à
    // public name :string;
    // private religion: string
    //
    // constructor(name :string, religion: string) {
    //      this.name = name;
    //      this.religion = religion;
    // }

}

// ***** Types Interface ***** //

interface canWalk{};

interface Animal extends canDrink, canWalk {};

// ***** Paramètres optionnels ***** //

function learnFormationOption (person: hasSkills, formation?: Formation):void {}

// ***** Décorateurs ***** //

const Log = function () {
    return (target: any, name: string, descriptor: any) => {            // target : la
méthode ciblée par notre décorateur
        console.log(`call to ${name}`);                                 // name : le
nom de la méthode ciblée
        return descriptor; };                                           // descriptor
: le descripteur de la méthode ciblée,
};

class Kid {

    @Log()
    sayHello (){}

    @Log()
    sayByeBye (){}

}
```

9.2. Typescript config sample

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",                          /* Specify ECMAScript target version:
'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'. */
    "module": "commonjs",                     /* Specify module code generation:
'commonjs', 'amd', 'system', 'umd' or 'es2015'. */
    // "lib": [],                             /* Specify library files to be included
in the compilation:  */
    // "allowJs": true,                       /* Allow javascript files to be
compiled. */
    // "checkJs": true,                       /* Report errors in .js files. */
    // "jsx": "preserve",                     /* Specify JSX code generation:
'preserve', 'react-native', or 'react'. */
    // "declaration": true,                   /* Generates corresponding '.d.ts' file.
*/
    "sourceMap": true,                        /* Generates corresponding '.map' file.
*/
    // "outFile": "./",                       /* Concatenate and emit output to single
file. */
    // "outDir": "./",                        /* Redirect output structure to the
directory. */
    // "rootDir": "./",                       /* Specify the root directory of input
files. Use to control the output directory structure with --outDir. */
    // "removeComments": true,                /* Do not emit comments to output. */
    // "noEmit": true,                        /* Do not emit outputs. */
    // "importHelpers": true,                 /* Import emit helpers from 'tslib'. */
    // "downlevelIteration": true,            /* Provide full support for iterables in
'for-of', spread, and destructuring when targeting 'ES5' or 'ES3'. */
    // "isolatedModules": true,               /* Transpile each file as a separate
module (similar to 'ts.transpileModule'). */

    /* Strict Type-Checking Options */
    "strict": true,                           /* Enable all strict type-checking
options. */
    // "noImplicitAny": true,                 /* Raise error on expressions and
declarations with an implied 'any' type. */
    // "strictNullChecks": true,              /* Enable strict null checks. */
    // "noImplicitThis": true,                /* Raise error on 'this' expressions
with an implied 'any' type. */
    // "alwaysStrict": true,                  /* Parse in strict mode and emit "use
strict" for each source file. */

    /* Additional Checks */
    // "noUnusedLocals": true,                /* Report errors on unused locals. */
    // "noUnusedParameters": true,            /* Report errors on unused parameters.
```

```
*/
    // "noImplicitReturns": true,             /* Report error when not all code paths
in function return a value. */
    // "noFallthroughCasesInSwitch": true,    /* Report errors for fallthrough cases
in switch statement. */

    /* Module Resolution Options */
    // "moduleResolution": "node",            /* Specify module resolution strategy:
'node' (Node.js) or 'classic' (TypeScript pre-1.6). */
    // "baseUrl": "./",                       /* Base directory to resolve non-
absolute module names. */
    // "paths": {},                           /* A series of entries which re-map
imports to lookup locations relative to the 'baseUrl'. */
    // "rootDirs": [],                        /* List of root folders whose combined
content represents the structure of the project at runtime. */
    // "typeRoots": [],                       /* List of folders to include type
definitions from. */
    // "types": [],                           /* Type declaration files to be included
in compilation. */
    // "allowSyntheticDefaultImports": true,  /* Allow default imports from modules
with no default export. This does not affect code emit, just typechecking. */

    /* Source Map Options */
    // "sourceRoot": "./",                    /* Specify the location where debugger
should locate TypeScript files instead of source locations. */
    // "mapRoot": "./",                       /* Specify the location where debugger
should locate map files instead of generated locations. */
    // "inlineSourceMap": true,               /* Emit a single file with source maps
instead of having a separate file. */
    // "inlineSources": true,                 /* Emit the source alongside the
sourcemaps within a single file; requires '--inlineSourceMap' or '--sourceMap' to be
set. */

    /* Experimental Options */
    "experimentalDecorators": true,          /* Enables experimental support for ES7
decorators. */
    "emitDecoratorMetadata": true            /* Enables experimental support for
emitting type metadata for decorators. */
  }
}
```