Angular

Déroulé du cours :
Jour 1 : révision JS et découverte des "web component"
Jour 2 : découverte TypeScript
Jour 3 : Installation et découverte Angular et des principaux "design pattern" qu'il utilise à travers la mise en place d'une "ToDoList"
Jour 4 : Découverte de la programmation réactive avec RXJS
Jour 5 : Finalisation du TP "todoList" avec l'injection de service pour la consommation d'un Observable avec httpClient
Jours 6, 7, : TP
Jour 7 : Tests unitaires
Jour 8, 9 et 10 TP noté

1 Introduction

Angular est un framework Javascript qui suit les principes suivants :

- la construction des applications est orientée "component", c'est à dire qu'une application est un assemblage de composants
- les composants Angular reposent sur les standards du Web (Web Component)
- un composant Angular est composé de 2 éléments minimum :
    - un fragment HTML (la vue)
    - une classe TypeScript (la logique)
- Le DATA binding :
    - Liaisons de données uni-directionnelle: .
        - Interpolation --> ((myVar)) (js --> html) .
        - Liaison de propriéte --> lattrl="myVar" (js --> html) •

        - Liaison d'événement --> (event) (html --> js)
        - Liaison de classe --> [ngClass] ou [class.‹className>1 • Liaison de style --> [ngStyle] ou [style.] -
- Liaison de données bi-directlonnelle
  Par ailleurs Angular
- utilise TypeScript qui apporte un typage fort et qui s'appuie sur le JavaScript moderne (ECMASCRIPT 2015 et +),
- fournit une Command Line Interface qui permet de gagner du temps pour :
    - la création d'un projet,
    - générer du code,
    - configurer des tests,
    - générer les livrables,
    - ...

1.1 Les principaux outils utilisés par Angular :

- NODE - Environnement d'exécution JS coté serveur
- NPM - Gestionnaire de paquets (dépendances)
- TypeScript - Surlangage (préprocesseur) et transpileur (compilateur source a source)

- SASS - Surlangage (preprocesseur) CSS et transpileur (compllateur source a source)
- WEBPACK - Bundler (permet de gérer plus facilement les imports et exports et de créer la version de production du projet via la commande 'ng build' et de lancer un serveur de développement via la commande 'ng serve')

2 ECMASCRIPT 2015+

2.1 let et const

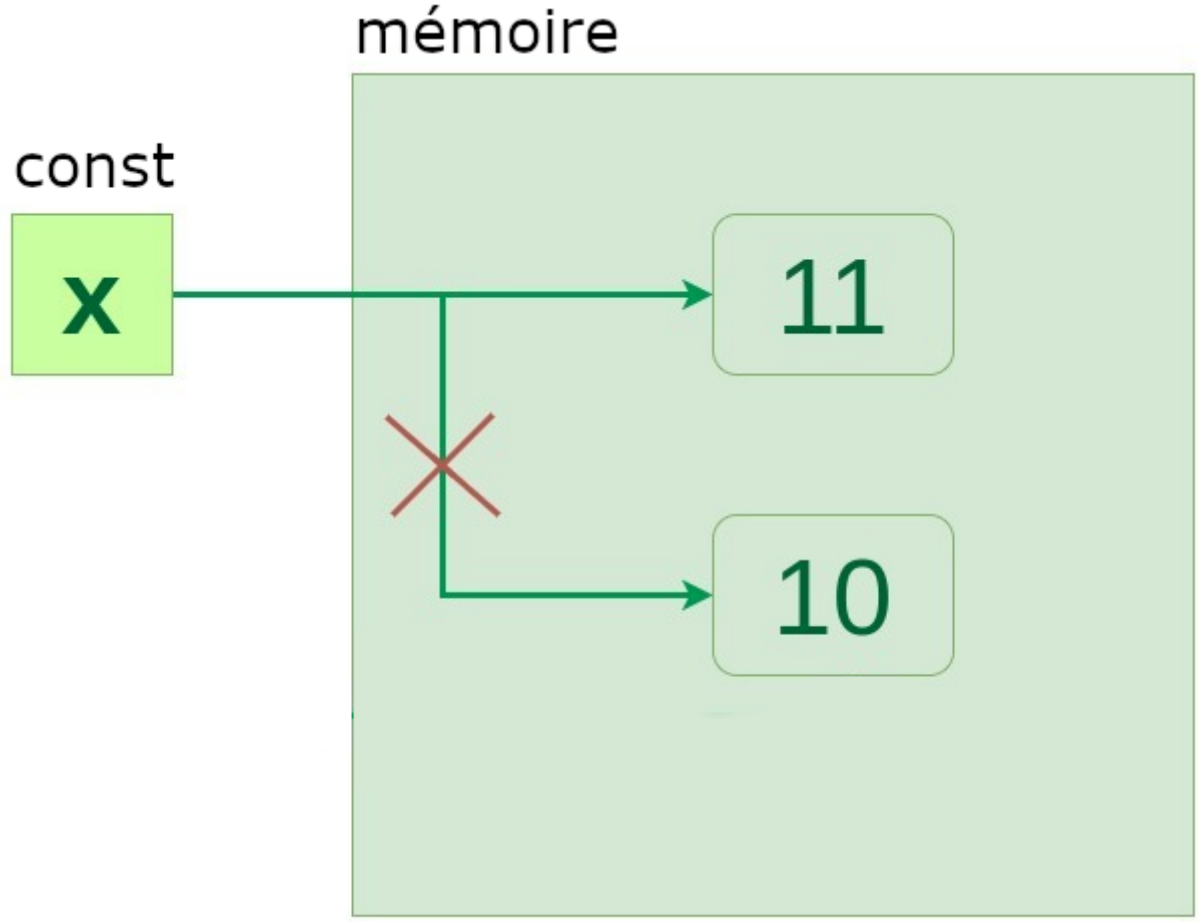
let et const sont les deux mots clés qui ont remplacé le mot clé var.

Les différences entre let, const et var :

- il n'y a pas de hoisting avec let et const
- les variables déclarées avec let et const sont "block scope" alors que les variables déclarées avec let sont "function scope"

Différence entre let et const : les variables déclarées avec const ne peuvent pas être réassignées.
Autrement dit, elles ne peuvent pas changer de référence.
En revanche leur valeur peut être modifiée dans le cas où

leur type est évolué.



2.2 Arrow function

Une expression de fonction fléchée (arrow function en anglais) permet d’avoir une syntaxe plus courte que les expressions de fonction et n'a pas le même mécanisme d'affectation de "this". Ce dernier prendra la valeur du contexte de création de la fonction. Si la fonction fléchée est créée dans le contexte global, "this" sera alors "window" en revanche si elle est créée à l'intérieur d'une classe, "this" prendra la valeur de l'instance en cours de la dite classe.

Ex :

```
let a = () => {
  console.log("Hello world");
}
a();
```

2.3 Raccourcis création d'objets

```
function createPerson() {
        const lastname = 'Dylan';
        const firstname = 'Bob';
        return { lastname: lastname, firstname:
firstname };
}
```

peut être simplifié en :

```
function createPerson() {  
        const lastname = 'Dylan';  
        const firstname = 'Bob';
        return { lastname, firstname };
}
console.log(createPerson());
```

Autre raccourci pour déclarer une méthode dans un objet :

```
function createPerson() {
        return {
                run: () => {
                        console.log('GO!');
                }
        };
}
```

qui peut être simplifié en :

```
function createPerson() {  
        return {  
                run() {
                console.log('GO!');  
                }  
        };
}
console.log(createPerson());
```

Explication : La fonction createPerson retourne un objet qui possède la méthode run.

2.4 Destructuring

2.4.1 Exemple pour les objets

```
const httpOptions = { timeout: 2000, isCache: true
};
```

```
const { timeout, isCache } = httpOptions;
console.log("timeout : ", timeout);
```

2.4.2 Exemple pour les tableaux

```
const timeouts = [1000, 2000, 3000];
const [shortTimeout, mediumTimeout] = timeouts;
console.log("shortTimeout : ", shortTimeout)
```

2.5 Paramètres optionnels et valeurs par défaut

```
function getCards(size = 10, page = 1) {  
        console.log("size : ", size);
}
getCards(50);
```

2.6 Rest Operator

Le rest operator peut être utilisé pour récupérer un **nombre variable de paramètres.**

```
const people = [];
function addPeople() {  
        for (var i = 0; i < arguments.length; i++)
{  
                people.push(arguments[i]);  
```

```
        }
}
addPeople('Simone', 'Germain');
console.log(people);
```

Peut se transformer en

```
const people = [];
function addPeople(...persons) {  
        for (let person of persons) {  
                people.push(person);  
        }
}
addPeople('Simone', 'Germain');
console.log(people);
```

... persons est maintenant un tableau sur lequel on peut itérer avec la boucle for...of

Le rest operator peut également fonctionner avec des affectations destructurées

```
const peopleInRace = ["Arlette", "José", "Bran"];
const [winner, ...losers] = peopleInRace;
console.log("winner : ", winner);
console.log("losers : ", losers);
```

Le **rest operator** ne doit pas être confondu avec le **spread operator** ("opérateur d'étalement"), même s'ils se

ressemblent. Le spread operator est son opposé : il prend un tableau, et l'étale en arguments variables.

```
const peopleInRace = ["Arlette", "José", "Bran"];
const people = ["Claudine", ...peopleInRace,
"Paulette"];
console.log("people : ", people);
```

2.7 Classes

Depuis ECMAScript ES6, il est possible de créer des classes d'objets avec un mécanisme d'héritage

Ex :

```
// Création d'une "class" Personne ES6
class Personne { // Majuscule selon les standards
  constructor(nom,prenom) { // récupération des
paramètres
    this.nom = nom; // propriété
    this.prenom = prenom; // propriété
  }
  // Méthodes ajoutées automatiquement au
prototype de Personne
  sePresenter() {
    console.log("Bonjour, je m'appelle " +
    this.prenom + " " + this.nom);
  }
}
```

```
/**
* instanciation d'une Personne avec passage
* des paramètres "Chazal" et "Franck" au
constructeur
*/
var franck = new Personne("Chazal","Franck"); //
franck.sePresenter();
// Création d'une "class" Enseignant qui hérite
// de la class Personne
class Enseignant extends Personne {
  constructor(nom,prenom,diplome) {
    super(nom,prenom);
    this.diplome = diplome;
  }
  // Méthodes
  sePresenter() {
&amp;nbsp; &amp;nbsp; super.sePresenter();&amp;nbsp;
&amp;nbsp; &amp;nbsp; console.log("... et je suis un
enseignant");
&amp;nbsp; }
  enseigner() {
    console.log("J'enseigne !");
  }
}
var jean = new
Enseignant("Dujardin","Jean","Agrégation");
jean.sePresenter();
jean.enseigner();
```

```
// Classe qui spécialise la class Enseignant
class EnseignantProgrammation extends Enseignant {
  // Méthodes
  enseignerJS() {
    console.log("J'enseigne le JS !");
  }
}
var yvan = new
EnseignantProgrammation("Attal","Ivan","BAC");
yvan.sePresenter();
yvan.enseignerJS();
```

2.7.1 Propriétés privées avec getter et setter

Depuis ECMAScript 2020 (ES11), il est possible de gérer des propriétés privées avec getter et setter.

Références :

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get

Exemple :

```
class Person {
  #name;
  constructor(name) {
    this.#name = name;
  }
  get name() {
    return this.#name;
  }
  set name(new_name) {
    this.#name = new_name;
  }
}
const b = new Person("Bob");
console.log(b.name);
b.name = "toto";
console.log(b.name);
```

2.7.2 Propriétés et méthodes de classe avec le mot clé static

cf
: https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes/static

Le mot-clé static permet de définir une propriété statique d'une classe. Les propriétés statiques ne sont pas disponibles sur les instances d'une classe mais sont appelées sur la classe elle-même. Les méthodes statiques

sont généralement des fonctions utilitaires (qui peuvent permettre de créer ou de cloner des objets par exemple).

```
class ClassWithStaticMethod {
  static staticProperty = 'someValue';
  static staticMethod() {
    return 'static method has been called.';
  }
  static {
    console.log('Class static initialization block
called');
  }
}

console.log(ClassWithStaticMethod.staticProperty);
// Expected output: "someValue"
console.log(ClassWithStaticMethod.staticMethod());
// Expected output: "static method has been
called."
```

2.8 Promesses

L'objet Promise , apparu avec ES2015; est utilisé pour réaliser des traitements de façon asynchrone. Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais !

Une promesse a 3 états :

- pending (en cours)
- resolve (résolue)
- reject (rejetée)
  Exemple de création et de consommation de promesses :

```
getToken = () => {
  return new Promise((res, rej) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        const token =
"qsdfEDLSoie5d8899;dEDd";
        console.log("Token
ok");
        res(token);
      } else
        rej(new Error("Pas de
chance, vous n'avez pas pu obtenir de token"));
    }, 2000);
  });
};

getUser = token => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        console.log("User ok",
```

```
token);
        resolve({ id: 1,
token: token });
      } else reject(new Error("Pas
d'utilisateur"));
    }, 2000);
  });
};

getToken()
  .then(value => {
    console.log("value dans le premier
then : ", value);
    // notez ici que "then" doit
renvoyer une promesse pour que l'on puisse
"chaîner"
    return getUser(value);
  })
  .then(value => {
    console.log("value dans le deuxième
then : ", value);
  })
  .catch(error => {
    console.error("Erreur: ",
error.message);
  });
```

2.9 Async/Await

La déclaration **async function** et le mot clé await sont des « sucres syntaxiques » apparus avec ES2017. Ils permettent de retrouver une syntaxe plus classique et donc plus lisibles.

2.9.1 async

Le mot clé **async** devant une déclaration de fonction la transforme en fonction asynchrone. Elle va retourner une promesse. Si la fonction retourne une valeur qui n'est pas une promesse, elle sera automatiquement comprise dans une promesse.

La promesse sera résolue avec la valeur renvoyée par la fonction asynchrone ou sera rompue s'il y a une exception non interceptée émise depuis la fonction asynchrone.

2.9.2 await

Le mot clé await est valable uniquement au sein de fonctions asynchrones définies avec async.
await interrompt l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue ou rejetée.

2.9.2.1 Exemple de code

```
// Promesse en utilisant async et await
async function getUniversities() {
  try {
    const response = await
fetch("http://universities.hipolabs.com/search?
country=Italy");
    const universities = await response.json();
    return universities;

  } catch (error) {
    console.error('Erreur attrapée : ', error);
  }
}
(async ()=>{
        const universities = await
getUniversities();
        console.log(universities);
})()
```

2.10 Map

Depuis ES2015, JS possède des collections.

```
const cedric = { id: 1, name: 'Cedric' };
const users = new Map();
users.set(cedric.id, cedric); // adds a user
console.log(users.has(cedric.id)); // true
console.log(users.size); // 1
```

```
console.log("users : ", users);
users.delete(cedric.id); // removes the user
console.log("users : ", users);
const josette = { id: 2, name: 'josette' };
users.set(josette.id, josette);
const rene = { id: 3, name: 'René' };
users.set(rene.id, rene);
console.log("users : ", users);
for (let user of users) {  
        console.log("user: ", user);
}
```

2.11 Template de string

Les littéraux de gabarits sont des littéraux de chaînes de caractères permettant d'intégrer des **expressions**, c'est-à-dire tout ce qui retourne une valeur.

L'usage de base consiste à imbriquer des variables dans les chaînes, entre **${ et }**. Elles se verront "remplacées" par leur valeur au moment de l'exécution.

```
let nb_kiwis = 3;
const message = `J'ai ${nb_kiwis} kiwis dans mon
panier`;
// Résultat : J'ai 3 kiwis dans mon panier
```

Exemple avec une fonction

```
function timestamp() { return new Date().getTime()
}
const message = `Le timestamp actuel est
${timestamp()}`;
```

2.12 Modules

Depuis ES6, on peut gérer les dépendances entre fichiers avec les mots clés "import" et "export"

Ex :

```
export default class Person {
  constructor(name) {
    this.name = name;
  }
  present() {
    console.log("hello, I'm " + this.name);
  }
}
```

De cette façon, dans un autre script, on pourra importer la classe Person :

```
import Person from "./Person.js";
const p = new Person("Bob")
```

Attention, il faudra penser à appeler votre js en utilisant l'attribut type="module"

```
<script type="module" src="test.10-module.js">
```

3 Web components

Le but des web composants est de fournir du code réutilisable et encapsulé directement utilisable comme une balise HTML classique.

Attention, ce n'est pas encore une norme et bon nombre de navigateurs n'interprèteront pas correctement le code ci-dessous qui repose sur 3 mécanismes :

- Custom elements
- Shadow DOM
- Template

3.1.1 Custom elements

C'est un nouveau standard qui permet de créer ses propres éléments du DOM comme

```
<digi-card></digi-card>
```

Exemple de déclaration d'un élément custom :

```
export default class DigiCardComponent extends
HTMLElement {
  constructor(){
    super();
    console.log(`Dans le constructeur de la
carte`);
  }
  /**
   * Appelé quand le composant est inséré
   */
  connectedCallback() {
    this.innerHTML = `<h2>Inventeur du web ?</h2>`
  }
}
customElements.define('digi-card',
DigiCardComponent);
```

3.1.2 Shadow DOM

Le problème dans l'exemple précédent, c'est que le contenu peut à tout moment être modifié par un script js.
Pour remédier à cela, on fait appel au "shadow DOM" :

```
export default class DigiCardComponent extends
HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode:
```

```
'open' });
    const question = document.createElement('h2');
    question.textContent = 'Inventeur du web ?';
    shadow.appendChild(question);
    console.log(`Dans le constructeur de la
carte`);
  }
}
customElements.define('digi-card',
DigiCardComponent);
```

Si on inspecte le code, on voit l'expression `#shadow-root` (open) qui indique que le "shadow DOM" fait bien office de barrière infranchissable.

3.1.3 Template

La balise "template" peut être directement utilisée dans un document HTML mais elle n'est pas affichée et rien de ce qui est déclaré à l'intérieur ne sera interprété par le navigateur. Pour que la balise template soit utilisable, Il faudra utiliser js en la clonant.

Ex de template dans le HTML :

```
<template id="card-template">
  <h2>Inventeur du web ?</h2>
```

```
  <h3>Tim Berners-Lee</h3>
</template>
```

... que l'on peut utilser dans le shadowDOM :

```
export default class DigiCardComponent extends
HTMLElement {
  constructor() {
    super();
    const template =
document.querySelector('#card-template');
    const clonedTemplate =
document.importNode(template.content, true);
    const shadow = this.attachShadow({ mode:
'open' });
    shadow.appendChild(clonedTemplate);

    console.log(`Dans le constructeur de la
carte`);
  }
}
customElements.define('digi-card',
DigiCardComponent);
```

3.1.4 ### Les limites des web components

Les web components n'étant pas supporté par tous les navigateurs, il existe le "polyfill" web-component.js pour s'assurer du bon fonctionnement.

4 TypeScript

Attention pour la compilation du TS fonctionne via Obsidian, il faut avoir installé "typescript" et "tsnode"

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

```
let i: number = 12;
console.log(i);
```

CF Cours sur TypeScript

5 Expressions vs instructions

Les expressions et les instructions (statements) présentent des différences.
Une expression sera évaluée plusieurs fois, par le mécanisme de détection de changement. Elle doit ainsi être la plus performante possible. Pour faire simple, une expression Angular est une version simplifiée d’une expression JavaScript.

Si on utilise user.name comme expression, user doit être défini, sinon Angular va lever une erreur.

Une expression doit être unique : on ne peut pas en enchaîner plusieurs séparées par des points-virgules.
Une expression ne doit pas avoir d'effets de bord. Par exemple, une affectation est interdite (ce n'est pas forcément le cas en js vanilla.

```
<!-- forbidden, as the expression is an
assignment, this will throw an error -->
<component [property]="user = 'Bob'"></component>
```

Elle ne doit pas contenir de mot clés comme if, var, etc.

De son côté, une instruction est déclenchée par l'événement correspondant.
Si on essaie d'utiliser une instruction comme task.show() où task serait undefined, cela déclenchera une erreur.

En revanche, on peut enchaîner plusieurs instructions, séparées par un point-virgule.
Une instruction peut avoir des effets de bord, et doit généralement en avoir : c'est l'effet voulu quand on réagit à un événement, on veut que des choses se produisent.
Une instruction peut contenir des affectations de variables, et peut contenir des mot-clés

6 Quelques notions de design Pattern

6.1 Injection de dépendance

cf https://en.wikipedia.org/wiki/Dependency_injection
L'injection de dépendances est une technique de programmation dans laquelle un objet ou une fonction reçoit d'autres objets ou fonctions dont il a besoin, au lieu de les créer en interne. L'injection de dépendances vise à séparer les préoccupations liées à la construction d'objets et à leur utilisation, conduisant à des programmes faiblement couplés.

Le modèle garantit qu'un objet ou une fonction qui souhaite utiliser un service donné n'a pas besoin de savoir comment construire ces services. Au lieu de cela, le « client » récepteur (objet ou fonction) reçoit ses dépendances par un code externe (un « injecteur ») dont il n'a pas connaissance.[

6.1.1 Premier exemple

Dans cet exemple classique, la classe "Car" dépend d'une interface Engine. Nous transmettons une instance qui implémente Engine au constructeur "Car" (c'est la partie Injection de dépendances). Nous utilisons ensuite une classe "Injector" pour gérer nos dépendances. Nous enregistrons

notre instance V8Engine auprès de l'injecteur, puis lorsque nous créons une voiture, nous utilisons l'injecteur pour résoudre les dépendances et les transmettons au constructeur de la voiture. De cette façon, la classe Car ne dépend pas directement de la classe V8Engine : la dépendance est injectée, ce qui rend le code plus modulaire et plus facile à tester et à gérer.

```
interface Engine { type: string; }

class Car {
&amp;nbsp; private engine: Engine;
&amp;nbsp; constructor(engine: Engine) {
&amp;nbsp; &amp;nbsp; this.engine = engine;
&amp;nbsp; }
&amp;nbsp; start(): void {
&amp;nbsp; &amp;nbsp; console.log(`Car with
${this.engine.type} engine started!`);
&amp;nbsp; }
}

class V8Engine implements Engine {
&amp;nbsp; type = 'V8';
}
class V12Engine implements Engine {
&amp;nbsp; type = 'V12';
}
```

```
// Injector
class Injector {
  // création d'un type via le type utilitaire "Record". Ici "dependencies" devient un objet qui a des clés de type string et des valeurs correspondantes de n'importe quel type
  private dependencies: Record<string, any> = {};
  // Méthode qui permet d'enregistrer une propriété de dépendencies sour la forme "clé/valeur"
  register(key: string, value: any): void {
    this.dependencies[key] = value;
  }

  // Méthode qui renvoie la valeur correspondant à la clé key de l'objet dependencies
  resolve<T>(key: string): T {
    return this.dependencies[key];
  }
}

// Register our dependencies
const injector = new Injector();
injector.register('engine', new V8Engine());
//injector.register('engine', new V12Engine());

// Resolve dependencies and create a new Car
const car = new Car(injector.resolve<Engine>
```

```
('engine'));

car.start(); // Outputs: Car with V8 engine
started!
```

Pour synthétiser ce qu'est l'injection de dépendance, prenons un composant d'une application, disons RaceList, permettant d'accéder à la liste des courses que le service RaceService peut retourner.

```
class RaceList {
        constructor() {
                this.raceService = new
RaceService();   // let's say that list() returns
a promise
                this.raceService
                .list()   // we store the races
returned into a member of `RaceList`
                .then(races => (this.races =
races));   // arrow functions, FTW!
        }
}
```

Mais ce code a plusieurs défauts. L'un d'eux est la testabilité : c'est compliqué de remplacer raceService par un faux service (un bouchon, un mock), pour tester notre composant.

Si nous utilisons le pattern d’injection de dépendance (Dependency Injection, DI), nous déléguons la création de RaceService à un framework, lui réclamant simplement une instance. Le framework est ainsi en charge de la création de la dépendance, et il peut nous "l’injecter", par exemple dans le constructeur :

```
class RaceList {
        constructor(raceService) {
                this.raceService = raceService;
                this.raceService.list().then(races
=> (this.races = races));
        }
}
```

Désormais, quand on teste cette classe, on peut facilement passer un faux service au constructeur :

```
// in a test
const fakeService = {
        list: () => {
        // returns a fake promise
        }
};
const raceList = new RaceList(fakeService);
// now we are sure that the race list
// is the one we want for the test
```

Mais comment le framework sait-il quel composant injecter dans le constructeur ? Bonne question !
C'est exactement ce que proposent les annotations de type : une métadonnée donnant un indice nécessaire au framework pour réaliser la bonne injection. En Angular, avec TypeScript, voilà à quoi ressemble notre composant RaceList :

```
class RaceList {
        raceService: RaceService;
        races: Array = [];
        constructor(raceService: RaceService) {
                // the interesting part is `: RaceService`
                this.raceService = raceService;
                this.raceService.list().then(races => (this.races = races));
        }
}
```

Maintenant l'injection peut se faire sans ambiguïté ! C'est pourquoi nous allons passer un peu de temps à apprendre TypeScript (TS). Angular est clairement construit pour tirer parti de celui-ci, et rendre notre vie de développeur plus facile en l'utilisant. Et l'équipe Angular a envie de soumettre le système de type au comité de standardisation, donc peut-être qu'un jour il sera normal d'avoir de vrais types en JS

6.1.2 Exercice

Imaginons que nous avons une class Product qui a pour propriétés privées :

- id
- name
- price

Imaginons que le calcul du prix dépend du type d'utilisateur.
Un utilisateur a pour propriétés privées :

- id
- login
  et pour méthode calculatePrice qui attend en paramètre un nombre et qui renvoie ce même nombre si c'est un utilisateur classique (User) et ce nombre * 0.9 si c'est revendeur (ResellerUser)

Créer deux utilisateurs (un User et un ResellerUser)
En utilisant un injecteur, créer un produit et afficher son prix
Constatez qu'en fonction du type d'utilisateur injecté, le prix ne sera pas le même pour le même produit.

Si vous avez le temps, faites en sorte que l'arboresence de votre application ressemble à ceci :
test-dependency-injection/
|

```
├── src/
│   ├── interfaces/
│   │   └── UserInterface.ts
│   ├── models/
│   │   ├── User.ts
│   │   ├── ResellerUser.ts
│   │   └── Product.ts
│   ├── utils/
│   │   └── Injector.ts
│   └── index.ts
│
└── package.json
```

6.2 Pattern observer

Le modèle Observateur (observer pattern) est un modèle de conception logicielle qui établit une dépendance un-à-plusieurs entre les objets de sorte que lorsqu'un objet change d'état, toutes ses dépendances sont notifiées et mises à jour automatiquement. Ce modèle est également appelé Publish-Subscribe ou Event-Subscriber pattern.

Voici quelques-uns des principaux avantages du modèle Observer :

Découplage d'objets : le modèle d'observateur vous permet de découpler ou de séparer des objets qui dépendent les

uns des autres. Dans ce schéma, le sujet et ses observateurs sont faiblement couplés. Le sujet n'a pas besoin de savoir quoi que ce soit sur les observateurs, seulement qu'ils implémentent l'interface Observer.

Relations dynamiques : ce modèle permet l'établissement dynamique de relations entre les sujets et les observateurs. Les observateurs peuvent être ajoutés et supprimés au moment de l'exécution sans modifier le sujet ou les autres observateurs.

Communication de diffusion : le modèle d'observateur permet à un sujet de diffuser des mises à jour à tous les observateurs enregistrés. Ceci est utile lorsqu'une modification apportée à un objet nécessite des modifications dans plusieurs autres.

Abstraction des composants principaux : le modèle Observer vous permet d'abstraire les composants principaux de votre application. Par exemple, dans une architecture Modèle-Vue-Contrôleur (MVC), le modèle (données) peut être le sujet observé et les vues (interfaces utilisateur) peuvent être les observateurs. Lorsque le modèle change, toutes les vues sont mises à jour automatiquement.

Dans votre code, la classe Subject représente le sujet et l'interface Observer représente les observateurs. La classe ConcreteObserver est une implémentation concrète de

l'interface Observer. La classe Subject gère une liste d'observateurs et fournit des méthodes pour ajouter (s'abonner), supprimer (se désinscrire) et notifier (notifier) les observateurs. L'interface Observer définit la méthode de mise à jour que les observateurs doivent implémenter. Cette méthode est appelée lorsque l'état du sujet change.

```
interface Observer {
  // propriété de type function. La fonction doit attendre un paramètre de n'importe quel type et ne renvoie rien
  update: (data: any) => void;
}

class Subject {
  // un tableau d'obervateurs
  private observers: Observer[] = [];

  // permet de souscrire à un observable, en fait, cela ajoute simplement un objet qui a au moins une méthode update au tableau "observers"

  subscribe(observer: Observer) {
     
this.observers.push(observer);
  }

// supprime un élément du tableau "observers"
```

```
  unsubscribe(observer: Observer) {
      const index =
this.observers.indexOf(observer);
      if (index > -1) {
         
this.observers.splice(index, 1);
      }
  }

// appelle la méthode update pour chacun des
observateurs
  notify(data: any) {
     
this.observers.forEach(observer =>
observer.update(data));
  }
}

// Une classe qui permet d'instancier des
observateurs
class ConcreteObserver implements Observer {
  update(data: any) {
     
console.log(`ConcreteObserver: Reacted to the
event with data: ${data}`);
  }
}

// création d'un nouveau sujet observable
```

```
const subject = new Subject();

// création d'un observateur
const observer = new ConcreteObserver();

// abonnement de l'observateur au sujet observable
subject.subscribe(observer);

// L'observable notifie une info à tous les
observateurs
subject.notify('Hello!');

// Désabonnement de l'observateurs à l'observable
subject.unsubscribe(observer);
```

6.3 Pattern singleton

Intérêt : Parfois, nous n'avons besoin que d'une seule instance de notre classe, par exemple une seule connexion à la base de données partagée par plusieurs objets, car la création d'une connexion à la base de données distincte pour chaque objet peut être coûteuse.

De même, il peut y avoir un seul gestionnaire de configuration ou un seul gestionnaire d'erreurs dans une application qui gère tous les problèmes au lieu de créer plusieurs gestionnaires. Le modèle singleton est un modèle de conception qui restreint l'instanciation d'une classe à un seul objet.

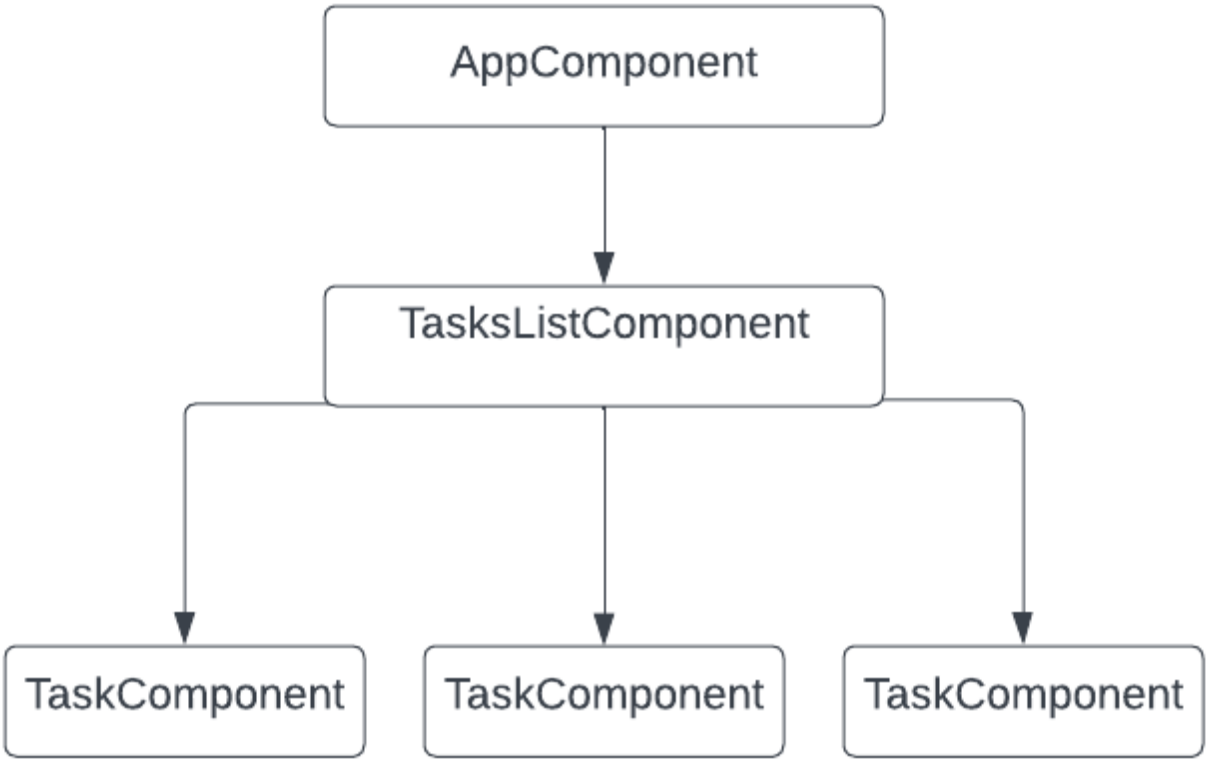
Exemple d'implémentation du pattern singleton :

```
class Singleton {
    private static instance: Singleton;

    private constructor() {
        // Initialization code here
    }

    public static getInstance(): Singleton {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }
}

// Usage
const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();

console.log(instance1 === instance2); // true
```

7 Les principaux concepts d'Angular

7.1 Composants

Angular est un framework qui utilise des composants comme React ou Vue. Un composant est un groupe d'éléments HTML dédiés à une tâche particulière.
Les composants Angular sont organisés hiérarchiquement comme le DOM.
Pour afficher une liste de tâches, il faudra créer un composant parent TaskList (par exemple) qui des tâches et des vues enfants pour chaque carte de tâche.



Un composant est une partie complètement isolée de l'application. L'application est un composant comme les autres. On regroupe les composants au sein d’une ou plusieurs entités cohérentes, appelées des modules (des

modules Angular, pas des modules ES2015), ou on se passe de ces modules en faisant des composants standalone.

7.2 Décorateurs

Comme nous l'avons vu avec TypeScript, il est possible d'utiliser des décorateurs et des annotations pour mieux définir le rôle des classes, des propriétés, des méthodes ou des paramètres.

7.3 Data binding

Le "Data Binding" dans Angular est une technique permettant de lier vos données à vos vues. En liant une variable, vous dites à Angular de surveiller les changements et de mettre à jour la vue chaque fois que les données changent et vice versa. Il existe quatre formes de liaison de données, qui diffèrent par la manière dont elles déplacent les données et le moment où elles sont généralement utilisées :

- Interpolation ({{ }}) :
  il s'agit d'une liaison de données unidirectionnelle du composant vers la vue. Il est utilisé pour incorporer des valeurs de chaîne dynamiques dans votre code HTML. Par exemple, si vous avez un titre de variable dans votre composant, vous pouvez l'afficher dans votre vue comme ceci :

```
<h1>{{ title }}</h1>
```

- Property Binding : il s'agit également d'une liaison de données unidirectionnelle du composant à la vue. Il est utilisé pour définir les propriétés des éléments de vue. Par exemple, vous pouvez désactiver un bouton comme celui-ci :

```
 <button [disabled]="isDisabled">Cliquez sur moi</button>.
```

- Event binding : il s'agit d'une liaison de données unidirectionnelle de la vue au composant. Il est utilisé pour gérer les événements générés par la vue. Par exemple, vous pouvez gérer un clic sur un bouton comme ceci :

```
<button (click)="handleClick()">Cliquez sur moi</button>.
```

- Two-Way binding ([(ngModel)]) : il s'agit d'une liaison de données bidirectionnelle, à la fois du composant à la vue et de la vue au composant. Il est utilisé avec les entrées de formulaire et d’autres éléments interactifs. Par exemple, vous pouvez vous lier à un champ de saisie comme celui-ci :

```
<input [(ngModel)]="name">.
```

Ces formes de liaison de données vous permettent de créer une expérience utilisateur dynamique et réactive en garantissant que votre vue reflète fidèlement vos données à tout moment.

7.4 Les "Zones" ou le concept de détection de changements

Angular utilise un concept appelé « Zones » pour décider quand effectuer la détection des changements. Une zone est un contexte d'exécution qui persiste y compris lorsque les tâches sont asynchrones. C'est comme un "local storage" et il vous permet de partager des données entre des fonctions asynchrones.

Angular s'exécute dans sa propre zone, connue sous le nom de **ngZone**. Cette zone est dérivée de la zone racine avec des comportements supplémentaires pour la détection des changements. Lorsque vous exécutez des tâches dans ngZone, Angular est informé de toute modification apportée aux "data bindings" et peut automatiquement mettre à jour le DOM pour refléter ces modifications.

Voici une explication simplifiée de son fonctionnement :

Lorsque vous démarrez votre application, Angular crée une nouvelle instance de NgZone, puis corrige toutes les API asynchrones (comme les événements setTimeout, Promise.then et DOM) pour garantir que toutes les opérations asynchrones sont exécutées dans la zone Angular.

Chaque fois que l'une de ces API asynchrones corrigées est appelée, NgZone est averti et déclenche une exécution de détection des modifications.

Angular vérifie ensuite toutes les liaisons de données dans l'application pour voir si certaines ont changé et met à jour le DOM pour refléter les modifications.

Une fois que toutes les tâches asynchrones sont terminées et qu'il n'y a plus de modifications, NgZone redevient stable.

Le processus réel est un peu plus complexe et implique des concepts tels que les microtâches et les macrotâches.

8 Première application et premier composant

8.1 Prérequis

- Avoir installé Node.js et NPM (vérifier que les versions sont récentes)

- Avoir installé Angular CLI. C'est un outil en ligne de commande pour démarrer rapidement un projet, déjà configuré avec Webpack comme un outil de construction, des tests, du packaging, etc.

8.2 Création de l'application avec ng

```
npm install -g @angular/cli@17.3.8
ng new todolist --prefix digi --defaults --
standalone
cd todolist
ng serve
```

Nous venons de créer un squelette d'application dans le dossier todolist. C'est depuis ce répertoire que l'on peut démarrer l'application avec **ng serve**. Cela démarre un serveur http local avec "hot loading"

9 Structure de l'application

9.1 Configuration

On peut commencer par s'intéresser aux fichiers de configuration :

- package.json : C’est là que sont définies les dépendances de l’application dont :

    - @angular
    - rxjs
    - zone.js
    - CLI, TypeScript, tests, ...
- tsconfig.json : fichier de config de TypeScript qui contient notamment :
    - experimentalDecorators pour pouvoir utiliser les décorateurs
    - sourceMap qui permet de générer les source maps ("dictionnaires de code source"), c’est-à-dire des fichiers assurant le lien entre le code JavaScript généré et le code TypeScript originel. Très utile pour débugger via le navigateur
- angular.json : fichier de configuration d'Angular CLI

9.2 Arborescence d'un projet Angular

/ (racine du projet)

- .angular/ --> Dossier de mise en cache
- .vscode/ --> Dossier de configuration concernant VSCode
- node_modules/ --> Dossier des dépendances et binaires du projet

- src --> Dossier contentant le coeur du projet
- .browserslistrc --> Liste des navigateurs (versions) pris en charge
- .editorconfig --> Configuration de l'éditeur
- .gitignore --> Fichiers ignores par Git
- angular.json --> Configuration de la cli d'Angular ('ng')
- karma.conf.js --> Configuration de Karma (framework de tests)
- package-lock,json --> Gel de l'arbre des dépendances du projet
- package.ison --> Métadonnées sur le projet, telles que le nom, la version, la description et l'auteur du projet. Comprend également des informations sur les dépendances, les scripts et autres configurations du projet.
- README,md --> Petite documentation du projet
- tsconfig.json --> Configuration du compilateur TypeScript
- tsconfig.app,json --> Extension de la configuration TS pour l'appIlcation
- tsconfig,spec.ison --) Extension de la configuration TS pour le tests
- /src

- app/ --> Dossier contenant toute la logique du projet
- assets/ --> Dossier réservé aux données brutes (images, pdf, audio...)
- environments/ --> Dossier contenant la constante d'environnement (en 2 fichiers disti ncts)
- favicon.lco --> lcône Favorite
- index.html --> Fichier chargé en premier par le navigateur
- main.ts --> Point d'entree du code source (défini le module racine du projet)
- polyfills,ts --> Code de rétrocompatibilite
- styles,css --> Styles globaux du projet
- test.ts --> Fichier d'initialisation de l'environnement de test

9.3 Composant

Un composant est la combinaison d'une vue (le template) et de logique (notre classe TS). La CLI a créé un composant : src/app/app.component.ts.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
```

```
@Component({
  selector: 'digi-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'todolist';
}
```

9.3.1 Convention de nommage

Il y a une convention de nommage clairement établie, et appliquée par Angular CLI. Les classes de composant se terminent par Component, et sont définies dans des fichiers dont le nom se terminent en .component.ts. Angular recommande aussi d'utiliser un préfixe dans les sélecteurs des composants, pour éviter un conflit de nom avec des composants externes. Par exemple, comme notre centre de formation se nomme Diginamic, on a choisi comme préfixe digi lors de la création de notre application en passant l'option --prefix digi. C'est ce que cette option permet : elle configure le projet afin que les composants soient générés avec le préfixe ns.

9.3.2 Décorateur @Component.

Notre application est elle-même un simple composant. Pour l'indiquer à Angular, on utilise le décorateur @Component. Pour pouvoir l'utiliser, il nous faut l'importer.

Il faut toujours penser à importer ce décorateur Component.

L'essentiel de nos besoins se situe dans le module @angular/core, mais ce n'est pas toujours le cas. Par exemple, quand on fera du HTTP, on utilisera des imports de @angular/http, ou quand on utilisera le routeur, on importera depuis @angular/router, etc.

9.3.3 Propriété selector

Elle indique à Angular ce qu'il faudra chercher dans nos pages HTML. À chaque fois qu'Angular trouve un élément dans notre HTML qui correspond au sélecteur défini dans notre composant, Angular crée une instance de ce composant, et remplace le contenu de l'élément par le template de notre composant.

Ici, chaque fois que notre HTML contiendra un élément :

```
<digi-root></digi-root>
```

... Angular créera une nouvelle instance de notre classe AppComponent.

9.3.4 Propriété templateUrl

Un composant doit aussi avoir un template. On peut avoir un template inline (directement dans le décorateur) ou dans un autre fichier comme le fait la CLI (ici app.component.html)
Le HTML correspondant est défini dans app.component.html, avec tout un tas d'éléments statiques et quelques éléments dynamiques comme :

```
<h1>Hello, {{ title }}</h1>
```

9.3.5 Propriété imports

```
imports: [CommonModule, RouterOutlet],
```

imports n'est pas toujours nécessaire. Son rôle est de donner à Angular la liste des autres composants, directives et pipes qui peuvent être utilisés à l'intérieur du template de notre composant. La plupart des composants que nous créons utilisent des pipes et des directives fournies par Angular.

Nous connaissons le principe des modules ES2015+ et TS dans qui permettent de définir des imports et des exports. si Angular rencontrait une balise dans le template de AppComponent, il faudrait qu'il sache où et comment est défini le composant correspondant.

Par exemple si nous souhaitons utiliser le composant TodoListComponent, nous devrons ajouter TodoListComponent aux imports du décorateur de AppComponent.

Les pipes et directives communément utilisés sont déclarés dans le module Angular CommonModule.

Importer CommonModule dans la configuration rend ainsi possible l'utilisation des pipes et directives de CommonModule dans le template de notre composant.

9.3.5.1 Directives et pipes de commonModule

CommonModule d'Angular est un module qui exporte toutes les directives et directives Angular de base, tels que NgIf, NgForOf, DecimalPipe, etc.

Les directives sont un moyen d'étendre la puissance du HTML en vous permettant de créer de nouveaux comportements ou composants. Ils sont l'une des principales

fonctionnalités d’Angular.
Une directive est une classe qui peut modifier la structure du DOM (Document Object Model) ou changer son comportement ou son apparence.

Par exemple, les directives NgFor et NgIf font partie de CommonModule et sont très utiles pour contrôler le rendu de parties du DOM

La directive NgIf est une directive structurelle qui vous permet d'inclure conditionnellement un modèle basé sur la valeur d'une **expression**. Cela revient à utiliser une instruction if dans un langage de programmation. Vous pouvez utiliser **NgIf** pour afficher de manière conditionnelle des parties de votre modèle, en fonction de certaines conditions

La directive **NgFor**, quant à elle, est utilisée pour restituer une liste ou une collection sur le DOM. C'est similaire à l'utilisation d'une **boucle for** dans un langage de programmation. Vous pouvez utiliser NgFor pour parcourir un tableau dans votre composant et créer un modèle pour chaque élément.

Une autre chose importante à noter est que CommonModule est automatiquement importé par BrowserModule lors de la création de l'application. Cela signifie qu’il n’est pas nécessaire d’importer explicitement le CommonModule à

utiliser. Les pipes et les directives seront importés automatiquement, afin qu'ils puissent être facilement utilisés

En résumé, les directives fournies par CommonModule sont très utiles car elles vous permettent de contrôler à la fois la structure et l'apparence du DOM dans vos applications Angular.

Par ailleurs et plus globalement, Il existe trois types de directives dans Angular :

- Directives de composants : ce sont des directives avec un modèle. Ce sont essentiellement des composants Angular
- Directives d'attribut : ces directives modifient l'apparence ou le comportement d'un élément, d'un composant ou d'une autre directive DOM. Par exemple, les directives intégrées NgStyle et NgClass peuvent modifier plusieurs styles ou classes à la fois.
- Directives structurelles : ces directives modifient la disposition du DOM en ajoutant et en supprimant des éléments du DOM. Par exemple, NgFor et NgIf sont des directives structurelles couramment utilisées.

Pour créer une directive, vous définissez une classe et la décorez avec @Directive(), en fournissant un sélecteur CSS qui identifie la directive lorsqu'elle est utilisée dans un modèle. Angular crée une nouvelle instance de la classe de

directive pour chaque élément correspondant, intégrant les capacités de la directive dans la vue.

9.3.6 Propriété standalone

```
standalone: true,
```

Cette propriété indique que notre composant est "standalone". Cela signifie que nous n’aurons pas besoin de le déclarer dans un module Angular pour pouvoir l’utiliser.

9.3.7 Propriété styleUrls

```
styleUrls: ['./app.component.css']
```

Cette propriété indique que notre composant utilise le fichier app.component.css pour la mise en page css.

9.3.8 Démarrage de l'application

L'application démarre avec l'application bootstrapApplication qui a été généré par Angular CLI dans main.ts. Ce fichier prend en argument le composant racine de l'application (AppComponent) :

```
import { bootstrapApplication } from
'@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from
'./app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

Le fichier **index.html** ne contient aucun appel à du javascript. C'est que l'application fonctionne dans un environnement orchestré par la CLI et qui utilise webpack pour créer des bundles (packages) et ajouter les scripts nécessaires. Vous remarquerez que le seul élément compris dans le body est l'élément racine **digi-root**. C'est dans cet élément que tout va s'afficher (ou presque) car nous sommes dans une SPA (Single Page Application).

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Todolist</title>
  <base href="/">
  <meta name="viewport" content="width=device-
width, initial-scale=1">
  <link rel="icon" type="image/x-icon"
href="favicon.ico">
```

```
</head>
<body>
  <digi-root></digi-root>
</body>
</html>
```

10 Les templates

Pour commencer, on peut faire un peu de ménage dans le template app.component.html créé automatiquement par la CLI. Vous voyez, que le fichier app.component.html embarque le style dans une balise ... cela frise la vulgarité.

On va tout simplement exporter ce code dans le fichier dédié aux css : app.component.css

La CLI a bien travaillé, elle a déjà donné la bonne valeur à la propriété styleUrls dans le fichier du composant racine app.component.ts

10.1 Interpolation

Le composant app.component.ts utilise déjà de l'interpolation puisque la propriété title qui est déclarée et assignée dans la class AppComponent.ts est ensuite utilisé (interpolée) dans le fichier de template :

```
<h1>Hello, {{ title }}</h1>
```

On comprend que ce sont les double-accolades qui indiquent que cette **expression** doit être évaluée. Si la valeur de la propriété "title" venait à être modifiée, le template sera automatiquement mis à jour grâce au mécanisme de "change detection". A noter : si on essaye d'afficher une variable qui n'est pas initialisée, au lieu d'afficher undefined, Angular affichera une chaîne vide, idem pour une variable null.

Par ailleurs **cette syntaxe est du sucre syntaxique pour l'écriture suivante :**

```
<h1 [textContent]="title"></h1>
```

La syntaxe à base de crochets permet de modifier la propriété textContent du DOM, et nous lui donnons la valeur title qui sera évaluée dans le contexte du composant courant, comme pour l'interpolation.
Notez que l'analyseur est sensible à la casse, ce qui veut dire que la propriété doit être écrite avec la bonne casse. textcontent ou TEXTCONTENT ne fonctionneront pas, il faut écrire textContent.

10.1.1 Binding de propriété

L’interpolation n’est qu’une des façons d’avoir des morceaux dynamiques dans nos templates. En fait, l’interpolation n’est qu’une simplification du concept au cœur du moteur de template d’Angular : le binding de propriété.

En Angular, on peut écrire dans toutes **les propriétés du DOM via des attributs spéciaux sur les éléments HTML, entourés de crochets** . Aussi étrange que cela puisse sembler, c’est du HTML valide . Le code ci-dessous est donc valide dans un template Angular :

```
<h1 [style]="{color:'red', fontSize:'2rem',
marginBottom:'30px'}">
  Hello, {{ title }}
</h1>
```

Cette syntaxe est très puissante et on peut l'utiliser dans beaucoup de situations comme par exemple :

```
<div [hidden]="isHidden">Hidden or not</div>
<digi-task [name]="task.fullName()"></digi-task>
```

Concernant le style, nous verrons plus tard qu'il existe une directive NgStyle.

Attribut vs propriété

Un nom d’attribut HTML peut commencer par n’importe quoi,

à l'exception de quelques caractères comme un guillemet ", une apostrophe ', un slash /, un égal =, un espace…

Prenons un exemple :

```
<input type="text" value="hello">
```

La balise input ci-dessus a deux attributs : un attribut type et un attribut value. Quand le navigateur rencontre cette balise, il crée un nœud correspondant dans le DOM (un HTMLInputElement), qui a les propriétés correspondantes type et value.

Chaque attribut HTML standard a une propriété correspondante dans un nœud du DOM. Mais un nœud du DOM a aussi d'autres propriétés, qui ne correspondent à aucun attribut HTML. Par exemple : childElementCount, innerHTML ou textContent.

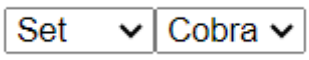
Les propriétés du DOM ont un avantage sur les attributs HTML : leurs valeurs sont forcément à jour. Dans notre exemple, l'attribut value contiendra toujours "hello", alors que la propriété value du DOM sera modifiée dynamiquement par le navigateur, et contiendra ainsi la valeur entrée par l'utilisateur dans le champ de saisie.

Enfin, les propriétés peuvent avoir des valeurs booléennes, alors que certains attributs n'agissent que par leur simple

présence ou absence sur la balise HTML. Par exemple, il existe l'attribut selected sur la balise : quelle que soit la valeur qui lui est donnée, il sélectionnera l'option, dès qu'il y est présent.
Ainsi le code suivant :

```
<select name="" id="">
        <option selected>Cobra</option>
        <option selected="false">Set</option> <!-- sélectionné -->
</select>

<select name="" id="">
  <option selected>Cobra</option>
  <option [selected]="false">Set</option> <!-- non sélectionné -->
</select>
```

ne donnera pas le même rendu pour les deux select :



10.2 Premiers exercices

Première tâche. Définissons maintenant une propriété plus complexe de type object qui détient les infos nécessaire à la description d'une tâche.

```ts
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'digi-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'todolist';
  task = {
    id: 1,
    name: 'Faire la vaisselle',
    done: false,
  };
}
```

Que l'on pourra simplement interpoler dans le template :

```html
<h3> {{ task.name }}</h3>
```

Puisqu'on y est et qu'il faut être réaliste, les tâches volent toujours en escadrille, continuons notre apprentissage en déclarant une variable tasks de type array :

```
...
export class AppComponent {
  title = 'todolist';
  tasks = [
    {
      id: 1,
      name: 'Faire la vaisselle',
      done: false,
      comment : "Dépêche toi mon lapin, je ne
supporte pas de voir traîner la vaisselle"
    },
    {
      id: 2,
      name: 'Faire le ménage',
      done: false,
    },
  ] | undefined;
}
```

Vous remarquerez que tasks peut-être indéfini et que la propriété comment peut ne pas exister.

✎ Exo

exo 1
Renseignez-vous sur les directive NgIf et NgFor et affichez les deux tâches du tableau "tasks" dans le template app.component.html en utilisant les balises

section, h2, ul et li. NgIf doit vous permettre d'afficher votre liste que dans le cas où la taille du tableau "tasks" est supérieure à 0. Vous utiliserez la syntaxe appropriée pour que l'absence éventuelle de commentaire ne génère pas d'erreur. Visuellement, le résultat devrait ressembler à cela :

10.3 Directive NgFor et NgIf

Vous êtes sans doute arrivé à produire un code s'approchant de cela :

```
<section *ngIf="tasks.length > 0">
        <h2>Les tâches qu'il te reste à faire :
</h2>
        <ul>
        <li *ngFor="let task of tasks">
```

```
            <div style="display: flex; gap: 20px">
              <span>{{ task.name }}</span>
              <span>{{ task?.comment }}</span>
            </div>
          </li>
      </ul>
</section>
```

Vous voyez que le "optional chaining operator" est bien pratique.

Dans l'exemple ci-dessus l'utilisation de NgIf et NgFor est possible grâce à l'importation de CommonModule. On aurait pu ajouter NgIf et NgFor au tableau imports sans importer CommonModule, c'est une question de préférence.

Vous remarquerez l'emploi de l'étoile (*) avant ngFor.
C'est une "microsyntaxe" qui est un "domain-specific language" (DSL) utilisé dans une « directive structurelle » de templae Angular pour guider avec une syntaxe très brève le comportement de la directive.

Sans cette microsyntaxe, notre code aurait ressemblé à :

```
<section *ngIf="tasks.length > 0">
   <h2>Les tâches qu'il te reste à faire :</h2>
   <ng-template ngFor task [ngForOf]="tasks" let-
i="index">
     <li *ngFor="let task of tasks">
```

```
        <div style="display: flex; gap: 20px">
          <span>{{ task.name }}</span>
          <span>{{ task?.comment }}</span>
        </div>
      </li>
    </ng-template>
  </section>
```

Par ailleurs, ngFor peut s'utiliser de la manière suivante :

```
<li *ngFor="let task of tasks; index as i">{{ i }}
- {{ task.name }}</li>
```

où index est une variable exportée. Certaines directives exportent des variables que l'on peut affecter à une variable locale comme :

- even, un booléen qui sera vrai si l'élément a un index pair
- odd, un booléen qui sera vrai si l'élément a un index impair
- first, un booléen qui sera vrai si l'élément est le premier de la collection
- last, un booléen qui sera vrai si l'élément est le dernier de la collection

10.3.1 Variable locale

Angular a la possibilité d'aller chercher les variables soit dans l'instance d'un composant, soit dans les variables locales. Les variables locales sont des variables que l'on déclare dynamiquement dans un template avec la notation "#".

Les variables locales sont aussi appelée variable de référence de modèle. Une variable de référence de modèle est souvent une référence à un élément DOM dans un modèle. Il peut également être utilisé pour faire référence à une instance d'une directive, un composant Web ou un template.

Voici un exemple d'utilisation d'une variable de référence de modèle :

```
<input #lastname type="text">
<button
(click)="logValue(lastname.value)">Afficher le
nom</button>
```

Un autre cas d’usage des variables locales est l’exécution d’une action sur un autre élément. Par exemple, on peut vouloir donner le focus à un élément quand on clique sur un bouton.

La méthode focus() est standard dans l’API DOM :

```
<input type="text" #name>
<button (click)="name.focus()">Focus the
input</button>
```

Cela peut aussi être utilisé avec un composant spécifique, créé dans notre application, ou importé d'un autre projet, ou même un véritable Web Component :

```
<google-youtube #player></google-youtube>
<button (click)="player.play()">Play!</button>
```

Ici le bouton peut lancer la lecture de la vidéo sur le composant . C'est bien un véritable Web Component écrit en Polymer ! Ce composant a une méthode play() qu'Angular peut appeler quand on clique sur le bouton.

10.3.2 ng-template

Les directives structurelles nous permettent de modifier la structure du DOM (ajouter des noeuds par exemple). Une directive est comme un composant mais sans template. Les directives structurelles fournies par Angular s'appuient sur l'élément ng-template , inspirée de la balise standard template de la specification HTML.
Un exemple de ng-template :

```
<ng-template>
        <h2>Tâches restantes</h2>
</ng-template>
```

Ce template ne sera pas affiché par le navigateur conformément à la spécification HTML. Cependant, si nous l'ajoutons dans une de nos vues, Angular pourra utiliser son contenu notamment via les directives structurelles. Par exemple :

```
<ng-template [ngIf]="tasks.length > 0">
  <h2>Tâches restantes</h2>
</ng-template>
```

Ici, le template ne sera instancié que si tasks a au moins un élément. Comme cette syntaxe est un peu longue, il y a une version raccourcie :

```
<h2 *ngIf="tasks.length > 0">Tâches restantes</h2>
```

La notation utilise "*" pour montrer que c’est une instantiation de template. La directive ngIf va maintenant prendre en charge l’affichage ou non de la div à chaque fois que la valeur de races va changer : s’il n’y a plus de course, la div va disparaître.

On utilisera toujours cette syntaxe courte.

10.4 ngClass

L'intérêt principal est que l'on peut changer plusieurs classes en même temps et faire dépendre la valeur des classes de n'importe quelle valeur booléenne

```
<li *ngFor="let task of tasks" [ngClass]="{'text-decoration-line-through': task.done, 'task-not-done': !task.done}">
```

10.5 ngStyle

L'intérêt principal est que l'on peut changer plusieurs styles en même temps .

```
<div [ngStyle]="{fontWeight: fontWeight, color: color}">I've got style</div>
```

La clé peut être en camelCase (fontWeight) ou en dash-case ('font-weight').

10.6 Pipe

Souvent, les données brutes n’ont pas la forme exacte que l’on voudrait afficher dans la vue. On a envie de les transformer, les formater, les tronquer, etc.

Comme les composants, les pipes peuvent être standalone ou pas. Les pipes fournis par Angular et dont nous allons discuter ici sont tous standalone, et font tous partie de CommonModule. Ainsi, pour pouvoir les utiliser dans un composant, il faudra les ajouter aux imports de ce composant, ou bien y ajouter le CommonModule.

10.6.1 json

Très utile en développement pour afficher la représentation json d'un objet

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

On peut même combiner les pipes :

```
<p>{{ tasks| slice:0:1 | json }}</p>
```

10.6.2 slice

Pour n'afficher qu'un sous-ensemble d'une collection.
Il fonctionne comme la méthode du même nom en JavaScript, et prend deux paramètres : un indice de départ et, éventuellement, un indice de fin. Pour passer un paramètre à un pipe, il faut lui ajouter un caractère ":" et le

premier paramètre, puis éventuellement un autre : et le second argument, etc. Ex :

```
<p>{{ tasks| slice:0:1 | json }}</p>
```

Cela fonctionne aussi avec les chaînes de caractères :

```
<h1 >{{ data | slice:0:3 }}</h1>
<h1 >{{ data | slice:-5 }}</h1>/* Affiche les 5
derniers caractères */
```

Il est également possible de stocker le résultat du slice dans une variable via la syntaxe "as".

10.6.3 async

Le pipe async permet d'afficher des données obtenues de manière asynchrone. Il peut gérer des données qui viennent d'une Promise ou d'un Observable.

Un pipe async retourne null jusqu'à ce que les données deviennent disponibles (i.e. jusqu'à ce que la promise soit résolue, dans le cas d'une promise).

Une fois résolue, la valeur obtenue est retournée. Et plus important, cela déclenche un cycle de détection de changement une fois la donnée obtenue.

Encore plus intéressant, si la source de données est un Observable, alors le pipe se chargera de se désabonner de la source de données à la destruction du pipe (quand l'utilisateur naviguera vers une autre page, par exemple, ou s'il fait partie du ngIf d'un template qui devient faux).

Un exemple de composant utilisant le pipe async :
Fichier ts :

```
import { Component } from '@angular/core';
import { Observable, interval } from 'rxjs';

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

export class AppComponent {
 data$: Observable<number>;
 constructor() {
    this.data$ = interval(1000);
 }
}
```

Fichier html

```
<div>{{ data$ | async }}</div>
```

10.6.4 Autres pipes

Angular fournit plus d'une dizaine de pipe dont voici une liste sans doute non exhaustive :

- CurrencyPipe : transforme un nombre en chaîne monétaire.
- DatePipe : transforme une date dans un format spécifié.
- DecimalPipe : Transforme un nombre en chaîne avec un point décimal.
- I18nPluralPipe : transforme un nombre en une pluralisation spécifique aux paramètres régionaux.
- I18nSelectPipe : transforme une chaîne en une version i18n spécifique aux paramètres régionaux.
- KeyValuePipe : transforme un objet ou une carte en un tableau de paires clé-valeur.
- LowerCasePipe : transforme le texte en minuscules.
- PercentPipe : transforme un nombre en chaîne de pourcentage.
- TitleCasePipe : transforme le texte en casse du titre.
- UpperCasePipe : transforme le texte en majuscules.

10.7 Bootstrap

Pour rendre notre interface plus propre, on peut utiliser bootstrap.

Pour cela, il faut d'abord l'installer :

```
npm install bootstrap
```

Puis ajouter la référence vers le fichier css de bootsrap dans le fichier angular.json :

```
"styles": [

"node_modules/bootstrap/dist/css/bootstrap.min.css
",
    "src/styles.css"
 ],
```

Redémarrer le server Angular

```
ng serve
```

Cela n'inclut que le CSS de Bootstrap, pas ses composants JavaScript. Si vous souhaitez utiliser les composants JavaScript de Bootstrap (comme les modaux, les info-bulles, etc.), vous pouvez envisager d'utiliser une bibliothèque comme ngx-bootstrap qui réimplémente le JavaScript de Bootstrap dans Angular.

10.7.1 Interface

Attention si vous gardez le code tel quel, il génèrera une erreur si votre tableau est vide. En effet le template ne comprendra pas que l'on utilise des propriétés (task.name par exemple) qui ne soient pas définies. Vous pouvez alors utiliser les interfaces TypeScript. Si l'on pense que l'interface en question va être utilisée dans plusieurs composants, alors voici la structure arborescente recommandée :

```
src/
|-- app/
|   |-- shared/
|   |   |-- interfaces/
|   |   |   |-- task.interface.ts
|   |-- component1/
|   |   |-- component1.component.ts
|   |-- component2/
|   |   |-- component2.component.ts
```

... ainsi que la définition de l'interface pour task :

```
export interface Task {
        id: number;
        name: string;
        done: boolean;
        comment?: string;
}
```

10.8 Evénements

Le navigateur déclenche des événements que l'on peut écouter : click, keyup, mousemove, etc.

Rajoutons un bouton "Valider" pour indiquer qu'une tâche est réalisée.

```
<button
(click)="onButtonValidate()">Valider</button>
```

... ce qui sous-entend que l'on a créé la méthode onButtonValidate dans le composant :

```
export class AppComponent {
  title = 'todolist';
  tasks: Task[] = [
    {
      id: 1,
      name: 'Faire la vaisselle',
      done: true,
      comment : "Dépêche-toi mon lapin, je ne
supporte pas de voir traîner la vaisselle"
    },
    {
      id: 2,
      name: 'Faire le ménage',
      done: false,
    },
```

```
  ];
  onButtonValidate():void {
    console.log("bouton validé cliqué");
  }
}
```

Il va ensuite falloir modifier la propriété tasks.

Exo

exo 2
Renseignez-vous sur la méthode map de js que vous utiliserez pour modifier le tableau "tasks"

Corrigé ? ›

Dans le template

```
 <button (click)="onButtonValidate(task.id)"
class="btn btn-success">Valider</button>
```

Dans le composant

```
onButtonValidate(taskId: number): void {
  this.tasks = this.tasks.map((task) => {
        if (task.id == taskId) task.done =
!task.done;
          return task;
```

```
        });
    }
```

Exo

Suite exo 2
... et faites en sorte que l'intitulé du bouton alterne entre "Valider" et "Invalider" et que la classe alterne entre btn-success btn-warning

Corrigé ?

A vous de vous débrouiller ! mais voici une piste : utiliser [ngClass] et [textContent].
Sinon, attendez la correction.

10.9 Génération d'identifiants unique

Vous verrez dans le code source des applications Angular des attributs du type : _ngconent -xxx

Il est ajouté par le mécanisme "View Encapsulation" d'Angular.

Angular utilise cet attribut pour appliquer des styles à des composants spécifiques sans affecter les autres éléments. Cela fait partie de la stratégie "Emulated View Encapsulation", qui est la stratégie par défaut utilisée par Angular.

Dans Angular, les styles CSS de chaque composant sont encapsulés dans la vue du composant et n'affectent pas le reste de l'application. Cela signifie que le CSS d'un composant spécifique n'interférera pas avec les autres composants. Cela se fait en ajoutant des attributs uniques aux éléments ainsi qu'aux règles CSS.

Par exemple, si vous avez un composant avec une règle CSS comme celle-ci :

```
p { color: blue; }
```

Angular va transformer cela en :

```
p[_ngcontent-xxx] { color: blue; }
```

Et il ajoutera l'attribut _ngcontent-xxx à tous les éléments p à l'intérieur du modèle du composant. De cette façon, la règle de style ne s'appliquera qu'aux p éléments faisant partie de ce composant.

Le "xxx" dans _ngcontent-xxx est un identifiant unique généré par Angular pour chaque composant.

10.10 Résumé

Le système de template d'Angular nous propose une syntaxe puissante pour exprimer les parties dynamiques de notre HTML. Elle nous permet d'exprimer du binding de données, de propriétés, d'événements, et des considérations de templating, d'une façon claire, avec des symboles propres :

- {{}} pour l'interpolation,
- [] pour le binding de propriété,
- () pour le binding d'événement,
- # pour la déclaration de variable,
- * pour les directives structurelles.

11 Créer des composants

Pour créer facilement un composant, la CLI Angular nous aide bien comme c'est d'ailleurs indiqué dans le README.md, il suffit d'exécuter la commande :

```
ng generate component nomDuComposant
```

Pour qu'un autre composant puisse utiliser ce nouveau composant, il faudra l'importer puis l'utiliser dans le template de la manière suivante :

```
import { TasksListComponent } from './nom-du-composant/nomDuComposant.component';
@Component({
  selector: 'digi-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, nomDuComposant],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',

})
```

```
<digi-nom-du-composant></digi-nom-du-composant>
```

11.1 Passage de paramètres

On transmets des données à un composant à l'aide du décorateur @Input().
Comme son nom le suggère, c'est ce décorateur qui va permettre la gestion d'entrants dans notre composant.

Supposons que vous ayez un composant parent et un composant enfant et que vous souhaitiez transmettre des

données du parent à l'enfant.

Dans votre composant enfant, déclarez une propriété @Input() :

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

@Component({ selector: 'app-child', template: `<p>
{{ data }}</p>`, })
export class ChildComponent {
        @Input() data: string = '';
}
```

Le décorateur @Input() ajoute des métadonnées à la classe qui rend la propriété appHighlight de la directive disponible pour la liaison entre le component parent et le composant enfant.
La propriété appHighlight est une directive d'attribut personnalisée dans Angular.
C'est donc grâce à elle que le la syntaxe suivante sera possible dans le template du composant parent :

```
<digi-tasks-list [data]="title"></digi-tasks-list>
```

Attention cependant à avoir mis en place les bons imports dans le composant parent :

```
import { ComposantEnfantComponent } from
'./composant-enfant/composant-enfant.component';

@Component({
  ...
  imports: [CommonModule, RouterOutlet,
ComposantEnfantComponent],

})

export class ParentComponent {
  ...
}
```

11.2 Cycle de vie d'un composant

> ⚠ **Warning**
>
> Il y a un point important à comprendre: les entrées d’un composant ne sont pas encore évaluées dans son constructeur.

Cela signifie qu’un composant comme celui-ci ne fonctionnera pas :

```
@Directive({
selector: '[undefinedInputs]',
```

```
standalone: true
})

export class UndefinedInputsDirective {
        @Input({ required: true }) person!:
string;
        constructor() {
                console.log(`inputs are
${this.person}`);
                // affichera tjs "inputs are
undefined"
        }
}
```

Pour accéder à la valeur d'une entrée, pour par exemple charger des données
complémentaires depuis un serveur, il faudra utiliser une phase du cycle de vie.
Il y a plusieurs phases accessibles, chacune avec ses spécificités propres :

- ngOnChanges sera la première appelée quand la valeur d'une propriété bindée est modifiée. Elle recevra une map changes, contenant les valeurs courante et précédente du binding, emballées dans un SimpleChange. Elle ne sera pas appelée s'il n'y a pas de changement.

- ngOnInit sera appelée une seule fois après le premier changement (alors que ngOnChanges est appelée à chaque changement). Cela en fait la phase parfaite pour du travail d'initialisation,
- ngOnDestroy est appelée quand le composant est supprimé. Utile pour y faire du nettoyage.

D'autres phases sont disponibles, mais pour des cas d'usage plus avancés :

- 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. Mais tu peux redéfinir cela si tu sais que la modification de certaines entrées n'a pas de conséquence. Cela peut être utile si tu veux accélérer le cycle de détection de changements en n'inspectant que le minimum, mais en règle générale tu ne devrais pas avoir à le faire.
- ngAfterContentInit est appelée quand tous les contenus projetés du composant ont été vérifiés pour la première fois.

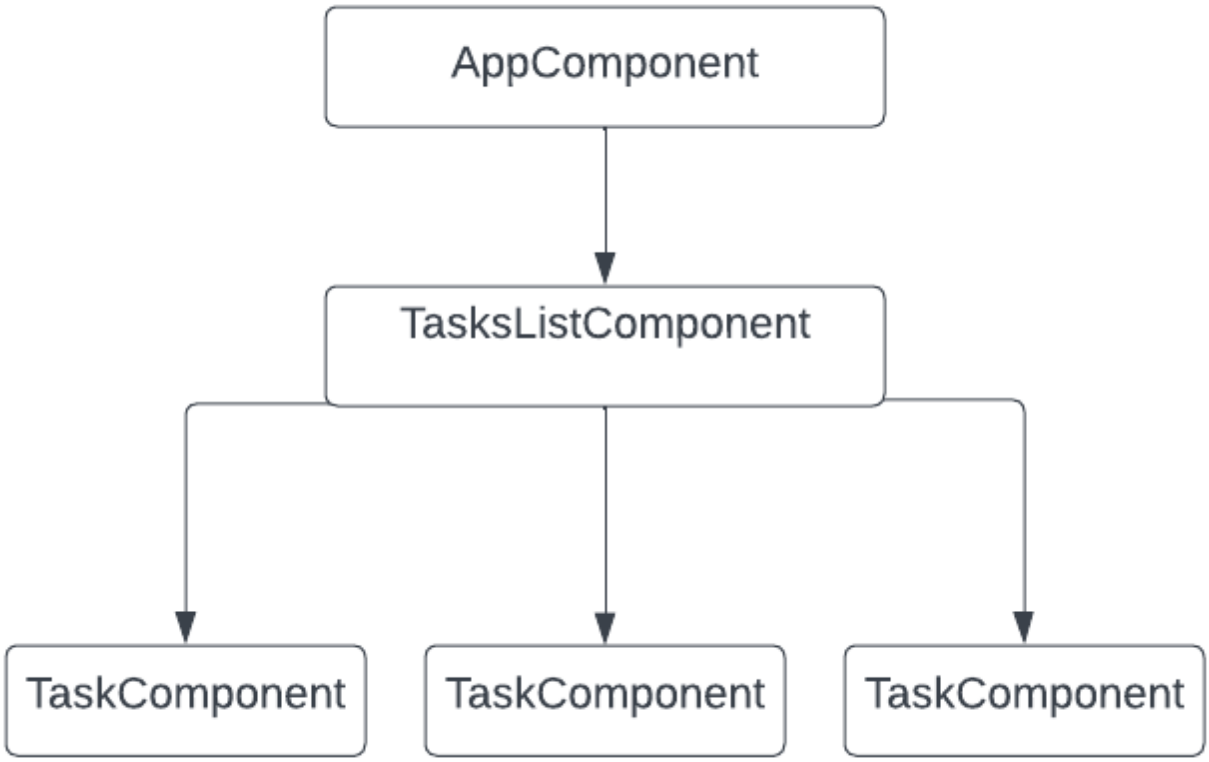
- ngAfterContentChecked est appelée quand tous les contenus projetés du composant ont été vérifiés, même s’ils n’ont pas changé.
- ngAfterViewInit est appelée quand tous les bindings du template ont été vérifiés pour la première fois.
- ngAfterViewChecked est appelée quand tous les bindings du template ont été vérifiés, même s’ils n’ont pas changé

✎ Exo

exo 3
Ecrivez un composant affichant

- un titre issu de la propriété title mais en le mettant en majuscule grâce à l'utilisation d'un pipe
- une liste de tâches (composant TasksList). Chacune de ces tâches sera représentée par son propre composant Task à qui l'on aura passé des paramètres

La représentation schématique de nos composants :



Ce que l'on attend visuellement :

Corrigé ?

A vous de vous débrouiller ! Si vous rencontrez des difficultés avec la gestion des événements, c'est normal, il

nous manque encore un peu de savoir...
La correction arrive !

Pour l'instant, on ne sait gérer des événements que dans la mesure où ils ont un impact sur une propriété du composant en question. Il va falloir apprendre à transmettre l'événement à un composant parent. Dans notre cas de figure, il faudrait que le click sur le bouton "Valider/Invalider" du composant Task ait un impact sur la propriétés "tasks" qui se trouve dans le composant TasksList. C'est le moment d'utiliser le "pattern Observer"

12 Programmation réactive

CF - cours RxJs

12.1 Introduction

La programmation réactive ou programmation fonctionnelle réactive (reactive programming) est devenu assez populaire dans la plupart des plateformes, comme en .Net avec la bibliothèque Reactive Extensions, qui est désormais disponible dans la plupart des langages (RxJava, RxJS, etc.).

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 tu croiseras 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, etc.

En javascript, on a déjà réagi à des événements en enregistrant des listeners sur des actions utilisateurs.

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 : on a pas à se soucier des conséquences d'un appel de méthode, on se contente 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, etc.

Angular est construit sur de la programmation réactive, et nous utiliserons aussi cette technique pour certaines parties :

- Répondre à une requête HTTP
- Lever un événement spécifique dans un de nos composants
- Gérer un changement de valeurs dans un de nos formulaires
- ...

12.2 Principe général

Dans la programmation réactive, tout est un flux. Un flux est une séquence ordonnée d'événements. Ces événements représentent :

- des valeurs ,
- des erreurs,
- des terminaisons.

Tous ces événements sont poussés par un producteur de données, vers un consommateur. En tant que développeur, il faudra s'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 a 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.

12.3 Un exemple d'observable issu d'un événement

Nous allons utiliser la bibliothèque RxJS pour créer un observable dans le composant enfant auquel le composant parent peut s'abonner. Cela permettra au composant parent de réagir aux événements qui se produisent dans le composant enfant.

Première étape
Dans votre composant enfant, créer un EventEmitter avec le décorateur @Output afin que le composant parent puisse s'y abonner. Par exemple, EventEmitter émettra des événements lorsque l'utilisateur clique sur le composant enfant.

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

@Component({
  selector: 'app-child',
  template: `<button (click)="onClick(1)">Click
me!</button>`
})
export class ChildComponent {
  @Output() clicked = new EventEmitter<number>();

  onClick(id:number) {
    this.clicked.emit(id);
  }
}
```

Deuxième étape

Attention à ne pas oublier de gérer l'événement "click" sur le bouton concerné dans le template du composant enfant :

```
...<button
  (click)="onClick(object.id)"
```

Troisième étape

Il faut que le composant parent s'abonne à EventEmitter Lorsque le composant enfant émettra un événement, la méthode handleClick du composant parent sera appelée :

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

@Component({
  selector: 'app-parent',
  template: `<app-child
(clicked)="handleClick($event)"></app-child>`
})
export class ParentComponent {
  handleClick(id: number) {
  console.log('Child component clicked with id:',
id);
}
```

Quelques explications :

@Output() est un décorateur qui marque un champ de

classe comme propriété de sortie. La propriété DOM liée à la propriété de sortie est automatiquement mise à jour lors de la détection des modifications.

@Output() est souvent utilisé en combinaison avec EventEmitter pour permettre aux données de circuler des composants enfants vers les composants parents. Le composant enfant émet un événement (à l'aide d'EventEmitter) et le composant parent écoute cet événement.

$event est un mot-clé spécifique à Angular qui fait référence aux données émises avec l'événement.

Exo

exo 3 suite
A l'aide des explications ci-dessus concernant la programmation réactive avec rxjs, faites en sorte que le click sur le bouton "Valider/Invalider" du composant Tasks émette un événement qui sera écouté par le composant TasksList pour modifier la propriété tasks

12.4 Récupération des données asynchrones avec rxjs

Classiquement la communication avec un serveur de données (souvent une API REST) se fait via un service dans les applications JS.
Angular n'échappe pas à la règle et il utilise 3 outils pour cela :

- l'injection de dépendance
- La programmation reactive avec RXJS
- le "pipe" async dans le template

12.4.1 Injection de dépendance

Pour faire de l’injection de dépendances, on a besoin :

- d’une façon d’enregistrer une dépendance, pour la rendre disponible à l’injection dans d’autres composants/services.
- d’une façon de déclarer quelles dépendances sont requises dans chaque composant/service. Le framework se chargera ensuite du reste.

Quand on déclarera une dépendance dans un composant, il regardera dans le registre s’il la trouve, récupérera l’instance existante ou en créera une, et réalisera enfin l’injection dans notre composant. Une dépendance peut être aussi bien un service fourni par Angular, ou un des services que nous avons écrit.

12.4.2 Utiliser un service fourni par Angular : Title

Admettons que nous voulions changer le titre de la page dès le composant parent App.

Il nous faudra d'abord importer le service "Title" puis l'injecter via le constructeur de notre composant :

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { Title } from '@angular/platform-browser';

import { TasksListComponent } from './tasks-
list/tasks-list.component';


@Component({
  selector: 'digi-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet,
TasksListComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',
})

export class AppComponent {
  constructor(headTitle: Title) {
```

```
    headTitle.setTitle('Ma toDoList de la mort');
  }
  title = 'Todo List !';
}
```

12.4.3 Créer un premier service synchrone

La CLI vient à notre aide :

```
ng generate service DataTasks
```

Modifions le code généré dans le fichier app/data-tasks.service.ts afin de créer la méthode loadTasks qui nous permet de récupére les tâches depuis notre service :

```
import { Injectable } from '@angular/core';
import Task from './shared/task.interface';

@Injectable({
  providedIn: 'root',
})

export class DataTasksService {
  constructor() {}
  loadTasks(): Task[] {
    return [
      {
```

```
        id: 1,
        name: 'Faire la vaisselle',
        done: true,
        comment:
          'Dépêche-toi mon lapin, je ne supporte
pas de voir traîner la vaisselle !',
      },
      {
        id: 2,
        name: 'Faire le ménage',
        done: false,
      },
    ];
  }
}
```

Injectons notre service dans le composant src/app/tasks-list/tasks-list.component.ts

```
import { Component, Input } from '@angular/core';
import { CommonModule } from '@angular/common';
import Task from '../shared/task.interface';
import { TaskComponent } from
'../task/task.component';
import { DataTasksService } from '../data-
tasks.service';

@Component({
  selector: 'digi-tasks-list',
```

```
  standalone: true,
  imports: [CommonModule, TaskComponent],
  templateUrl: './tasks-list.component.html',
  styleUrl: './tasks-list.component.css',
})

export class TasksListComponent {
  tasks: Task[] = [];
  constructor(private DataTasksService:
DataTasksService) {
    this.tasks =
this.DataTasksService.loadTasks().sort((a: Task,
b: Task) => Number(a.done) - Number(b.done));
  }
  @Input() data: string = '';
  handleClickButtonValidate(taskId: number): void
{
    console.log(`Dans handleClickButtonValidate`);
    this.tasks = this.tasks
      .map((task) => {
        if (task.id == taskId) return { ...task,
done: !task.done };
        return task;
      })
      .sort((a: Task, b: Task) => Number(a.done) -
Number(b.done));
  }
}
```

12.4.4 Utiliser un service pour envoyer et recevoir des données via HTTP

12.4.4.1 Mettre en place un server d'API REST

Tout cela est bien beau mais dans la réalité, les données proviennent de serveur ... de données. Souvent des serveur d'API REST. Cela implique une gestion asynchrone et en JS, dans ce cas, on fait souvent appel à la méthode "fetch" qui renvoie une promesse.

Avec Angular, on fait appel à de la programmation réactive en utilisant la librairie rxjs. C'est l'arme absolue pour gérer les flux de données asynchrones.

Pour simuler un serveur d'API REST, nous utilisons [json-server]
Installation de json-server :

```
npm install -g json-server
```

Création du fichier db.json

```
{
  "tasks": [
    {
      "id": 1,
```

```
      "name": "Faire la vaisselle",
      "done": true,
      "comment": "Dépêche-toi mon lapin, je ne
supporte pas de voir traîner la vaisselle !"
    },
    {
      "id": 2,
      "name": "Faire le ménage",
      "done": false
    }
  ]
}
```

Attention à bien vous placer dans le répertoire qui contient le fichier db.json pour lancer le serveur :

```
json-server db.json
```

Le serveur doit vous renvoyer un message de ce style dans la console :

```
  \{^_^}/ hi!

  Loading db.json
  Done

  Resources
  http://localhost:3000/tasks
```

```
Home
http://localhost:3000
```

12.4.4.2 Exécuter une requête HTTP

Une des possibilités est l'API fetch, qui est une API standard fournie par les navigateurs. On peut parfaitement construire une application en utilisant fetch mais la plupart des développeurs Angular utilisent un service fourni par Angular : HttpClient.

Il faudra utiliser les classes et fonctions du package @angular/common/http. Pourquoi préférer ce service à fetch ? La réponse est simple : pour les tests.

Comme on le verra, le module Http permet de bouchonner ton serveur, et de retourner des réponses données. C'est vraiment, vraiment très utile. Le client HTTP utilise largement le paradigme de la programmation réactive.

Le module @angular/common/http propose un service nommé HttpClient que tu peux injecter dans n'importe quel constructeur. Ce service n'est pas disponible par défaut dans une application Angular. Il faut configurer l'application afin de pouvoir l'utiliser, en ajoutant un provider au démarrage de l'application.

Par défaut, le service HttpClient réalise des requêtes AJAX avec XMLHttpRequest. Il propose plusieurs méthodes, correspondant au verbes HTTP communs :

- get
- post
- put
- delete
- patch
- head
  Toutes ces méthodes retournent un objet Observable.

L’utilisation des Observables apporte plusieurs avantages, comme la possibilité d’annuler une requête, de la retenter, d’en composer facilement plusieurs, etc.

Commençons par récupérer les tâches disponibles . Pour charger les tâches, on enverra une requête GET sur une URL : ' http://localhost:3000/tasks' comme nous l'avons vu plus haut. Généralement, l’URL de base de tes appels HTTP sera stockée dans une variable ou un service, que l'on pourra facilement variabiliser selon l'environnement. Ou, si l’API REST est servie par le même serveur que l’application Angular, on peut simplement utiliser une URL relative : '/api/tasks

Pour pouvoir utiliser les services du client HTTP, nous allons configurer l'injecteur racine de notre application Angular en modifiant le fichier src/app/app.config.ts

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideHttpClient } from
'@angular/common/http';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {

  providers: [provideRouter(routes),
provideHttpClient()]

};
```

Ensuite il nous faut modifier le service (app/data-tasks.service.ts) afin d'injecter le service HttpClient :

```
import Task from './shared/task.interface';

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
```

```
})

export class DataTasksService {
  constructor(private http: HttpClient) {}

  loadTasks(): Observable<Task[]> {
    const url = 'http://localhost:3000/tasks';
    const params = { status: 'PENDING' };
    return this.http.get<Array<Task>>(url, {
params });
  }
}
```

Il ne nous reste plus qu'à modifier le composant (src/app/tasks-list/tasks-list.component.ts) pour souscrire à l'observable renvoyé par loadTasks

```
import { Component, Input } from '@angular/core';
import { CommonModule } from '@angular/common';
import Task from '../shared/task.interface';
import { TaskComponent } from
'../task/task.component';
import { DataTasksService } from '../data-
tasks.service';
import { Observable, of } from 'rxjs';

@Component({
  selector: 'digi-tasks-list',
  standalone: true,
```

```
  imports: [CommonModule, TaskComponent],
  templateUrl: './tasks-list.component.html',
  styleUrl: './tasks-list.component.css',
})

export class TasksListComponent {
  tasks: Task[] = [];
  // Encore une belle injection de dépendance !
  constructor(private dataTasksService:
DataTasksService) {
  }

  ngOnInit(): void {
  // Vla la souscription

this.dataTasksService.loadTasks().subscribe((tasks
:Task[]) => {
      this.tasks = tasks;
    });
  }

  @Input() data: string = '';
  handleClickButtonValidate(taskId: number): void
{
    console.log(`Dans handleClickButtonValidate`);
    this.tasks = this.tasks
      .map((task) => {
        if (task.id == taskId) return { ...task,
done: !task.done };
```

```
        return task;
      })
      .sort((a: Task, b: Task) => Number(a.done) -
Number(b.done));
  }

}
```

Pour rappel, ngOnInit sera appelée une seule fois après le premier changement (alors que ngOnChanges est appelée à chaque changement). Cela en fait la phase parfaite pour du travail d'initialisation.

13 Routes

Angular fournit le module "RouterModule" qui est optionnel et qu'il faut donc configurer pour qu'il soit opérationnel. Comme nous l'avions fait pour le client HTTP, il nous faut modifier le fichier src/app/app.config.ts pour fournir le routeur à l'application

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideHttpClient } from
'@angular/common/http';

import { routes } from './app.routes';
```

```
export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes),
provideHttpClient()]
};
```

Ensuite, il faut créer une configuration pour définir les associations entre les URLs et les composants. Cela se fait généralement dans un fichier dédié nommé src/app/app.routes.ts :

```
import { Routes } from '@angular/router';
import { TasksListComponent } from "./tasks-
list/tasks-list.component";
import { AboutComponent } from
'./about/about.component';

export const routes: Routes = [
  { path: '', component: TasksListComponent },
  { path: 'about', component: AboutComponent }
];
```

Cela sous entend que l'on a préalablement créé le composant About :

```
ng generate component About
```

... et que l'on a modifié le template src/app/app.component.html :

```
<main class="main container">
  <router-outlet></router-outlet>
</main>
```

On comprend, à la lecture du code ci-dessus, que le tag router-outlet permet d'inclure le composant qui correspond à la route définie dans le fichier app.routes.ts
Si vos routes fonctionnent c'est que vous avez bien importé la directive du routeur "RouterOutlet" dans le composant "AppComponent" :

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'digi-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',

})

export class AppComponent {
  constructor(headTitle: Title) {
    headTitle.setTitle('Ma toDoList de la mort');
```

```
  }
  title = 'Todo List !';
}
```

13.1 Navigation

Pour passer d'une route à l'autre, on peut créer des liens via la directive "RouterLink".
La directive RouterLink peut recevoir soit une constante représentant le chemin vers lequel on veut naviguer, soit un tableau de chaînes de caractères, représentant le chemin de la route et ses paramètres. Par exemple, dans le template de AppComponent si l'on veut ajouter un header comportant une zone de navigation, on peut écrire :

```
<header>
  <a routerLink="/" routerLinkActive="selected-
menu">Home</a>
  <a routerLink="/about"
routerLinkActive="selected-menu">About</a>
</header>
<main class="main container">
  <router-outlet></router-outlet>
</main>
```

Notez l'utilisation de la directive routerLinkActive qui ajoute automatiquement une classe si le chemin courant

correspond à la route.

Cela n'est possible que si vous avez importé ces directives dans le fichier app.component.ts :

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet, RouterLink,
RouterLinkActive } from '@angular/router';
import { Title } from '@angular/platform-browser';


@Component({
  selector: 'digi-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet,
RouterLink, RouterLinkActive],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',
})

export class AppComponent {
  constructor(headTitle: Title) {
    headTitle.setTitle('Ma toDoList de la mort');
  }
  title = 'Todo List !';
}
```

14 Formulaires

Il existe deux façons d'écrire des formulaires avec Angular :

- Formulaire piloté par le template en utilisant des directives dans le template. Très utile pour des formulaires simples.
- Formulaire piloté par le code : il faut alors écrire une description du formulaire dans le composant puis utiliser des directives pour lier le formulaire aux inputs du template. C'est un peu plus compliqué mais plus puissant notamment pour générer des formulaires dynamiquement.

14.1 FormControl et FormGroup

Un champ du formulaire, comme un input ou un select, sera représenté par un **FormControl**. C’est la plus petite partie d’un formulaire qui encapsule l’état du champ et sa valeur.

14.1.1 FormControl

Un FormControl a plusieurs attributs :

- valid : si le champ est valide, au regard des contraintes et des validations qui lui sont appliquées.

- invalid : si le champ est invalide, au regard des contraintes et des validations qui lui sont appliquées.
- errors : un objet contenant les erreurs du champ.
- dirty : false jusqu’à ce que l’utilisateur modifie la valeur du champ.
- pristine : l’opposé de dirty.
- touched : false jusqu’à ce que l’utilisateur soit entré dans le champ.
- untouched : l’opposé de touched.
- value : la valeur du champ.
- valueChanges : un Observable qui émet à chaque changement sur le champ.

FormControl propose aussi quelques méthodes comme hasError() pour savoir si le contrôle a une erreur donnée

14.1.2 FormGroup

Un `FormGroup` rassemble la valeur et l'état de validité d'un groupe d'instances `FormControl` en un seul objet, avec chaque nom de contrôle comme clé. Il calcule son statut en réduisant les valeurs de statut de ses enfants. Par exemple, si l'un des contrôles d'un groupe n'est pas valide, l'ensemble du groupe devient invalide.

FormGroup est l'un des quatre blocs de construction fondamentaux utilisés pour définir des formulaires dans Angular, avec FormControl, FormArray et FormRecord.

Lors de l'instanciation d'un FormGroup, transmettez une collection de contrôles enfants comme premier argument. La clé de chaque enfant enregistre le nom du contrôle.

FormGroup est destiné aux cas d'utilisation où les clés sont connues à l'avance. Si vous devez ajouter et supprimer des contrôles de manière dynamique, utilisez plutôt FormRecord.

Un FormGroup a les mêmes propriétés qu’un FormControl, avec quelques différences :

- valid : si tous les champs sont valides, alors le groupe est valide.
- invalid : si l’un des champs est invalide, alors le groupe est invalide.
- errors : un objet contenant les erreurs du groupe, ou null si le groupe est entièrement valide. Chaque erreur en constitue la clé, et la valeur associée est un tableau contenant chaque contrôle affecté par cette erreur.
- dirty : false jusqu’à ce qu’un des contrôles devienne "dirty".
- pristine : l’opposé de dirty.

- touched : false jusqu'à ce qu'un des contrôles devienne "touched".
- untouched : l'opposé de touched.
- value : la valeur du groupe. Pour être plus précis, c'est un objet dont les clé/valeurs sont les contrôles et leur valeur respective.
- valueChanges : un Observable qui émet à chaque changement sur un contrôle du groupe.

Un groupe propose les mêmes méthodes qu'un FormControl, comme hasError(). Il a aussi une méthode get() pour récupérer un contrôle dans le groupe.

14.1.3 Exemples d'utilisations d'une instance de FormGroup

Le code `src\app\newsletter\newsletter.component.html` suivant :

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

```
@Component({
  selector: 'app-newsletter',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './newsletter.component.html',
})

export class NewsletterComponent implements OnInit
{
  newsletterForm!: FormGroup;
  constructor(private fb: FormBuilder) {}
  ngOnInit() {
    this.newsletterForm = this.fb.group({
      email: ['', [Validators.required,
Validators.email]],
    });
  }

  onSubmit() {
    if (this.newsletterForm.valid) {
      console.log('Email soumis :',
this.newsletterForm.value.email);
    }
  }
}
```

va permettre de :

- créer une instance de `FormGroup` (ici `this.newsletterForm`) en appelant la méthode `group`

avec comme argument un objet dont la clé `email` est un tableau.
    - La première valeur de ce tableau est une chaîne de caractères vide ce qui veut dire que l'input correspondant sera vide par défaut
    - le second élément de ce tableau est un tableau des `validators` qui seront appliqués à cette instance de `FormControl` (ici `required` et `email`)
- gérer la soumission du formulaire via la méthode onSubmit.
  - si les validators sont positifs alors on affiche la value de l'email : `this.newsletterForm.value.email`
  Le code ci-dessus est à mettre en relation avec le code du `template` suivant :

```
<form [formGroup]="newsletterForm"
(ngSubmit)="onSubmit()">
  <input type="email" formControlName="email"
placeholder="Votre email" />
  <button type="submit"
[disabled]="!newsletterForm.valid">S'abonner</butt
on>
</form>
```

Vous comprenez que :

- l'attribut `formControlName` reprend la clé `email` définie dans ngOnInit() du fichier composant ``src\app\newsletter\newsletter.component.ts
- la propriété `disabled` est "bindée" avec la propriété `!newsletterForm.valid`

14.2 Ecrire un formulaire piloté par le template

Lorque l'on écrit un formulaire piloté par le template, il nous suffira d'écrier des directives dans notre template du formulaire et laisser le framework construire les instances de FormControl et FormGroup nécessaires.

Par exemple, la directive NgForm transforme l'élément

```
<form>
```

en sa version Angular.
Toutes ces directives sont incluses dans le module FormsModule. Nous devons donc l'importer dans chaque composant qui utilise un formulaire piloté par le template.

Ecrivons notre composant formulaire src/app/form-task/form-task.component.ts:

```ts
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';


@Component({
  selector: 'digi-form-task',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './form-task.component.html',
  styleUrl: './form-task.component.css',
})

export class FormTaskComponent {
  register(taskTitle: string) {
    console.log(`Formulaire soumis`, taskTitle);
  }
}
```

Ainsi que le template associé src/app/form-task/form-task.component.html

```html
<!-- On utilise la variable locale #taskForm -->
<!-- et on donne sa valeur à la méthode register -
->
<form (ngSubmit)="register(taskForm.value)"
#taskForm="ngForm"class="d-flex gap-3 align-items-
center">
```

```
  <label>Tâche : </label>
  <input name="task" ngModel>
  <button type="submit" class="btn btn-
success">Ajouter une tâche</button>
</form>
```

"ngForm" est une directive fournie par Angular pour suivre le statut et la valeur du formulaire.

Ceci est du binding uni-directionnel, si on met à jour le champ, le modèle sera mis à jour, mais mettre à jour le modèle ne mettra pas à jour la valeur du champ. On peut cependant créer un binding multi-directionnel avec la directive ngModel mais pour l'instant nous n'en avons pas besoin.

14.3 Ecrire un formulaire piloté par le code (Reactive form)

Les `Reactive Forms` sont une approche de gestion de formulaires en Angular où la logique du formulaire est définie dans le composant TypeScript plutôt que dans le template HTML. Cela offre plus de contrôle et de flexibilité pour la validation, la manipulation des données et la gestion des états du formulaire.

14.3.1 Exemple

Exemple simple de formulaire d'abonnement à une newsletter :

- fichier newsletter.component.ts :

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

@Component({
  selector: 'app-newsletter',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './newsletter.component.html',
})

export class NewsletterComponent implements OnInit
{
  newsletterForm!: FormGroup;
  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.newsletterForm = this.fb.group({
      email: ['', [Validators.required,
Validators.email]],
```

```
    });
  }
  onSubmit() {
    if (this.newsletterForm.valid) {
      console.log('Email soumis :',
this.newsletterForm.value.email);
      // Ici, vous appelleriez normalement un
service pour envoyer l'email au backend
    }
  }
}
```

- fichier newsletter.component.html :

```
<form [formGroup]="newsletterForm"
(ngSubmit)="onSubmit()">

  <input type="email" formControlName="email"
placeholder="Votre email" />

  <button type="submit"
[disabled]="!newsletterForm.valid">S'abonner</butt
on>

</form>
```

Explications :

1. Dans le composant, nous importons les modules nécessaires de `@angular/forms`.
2. Nous créons une propriété `newsletterForm` de type `FormGroup`. Nous l'utiliserons dans le `template`
3. Dans `ngOnInit()`, nous initialisons le formulaire avec `FormBuilder`. Nous définissons un seul champ 'email' avec un des validateurs disponible : email .
4. La méthode `onSubmit()` vérifie si le formulaire est valide avant de traiter les données.
5. Dans le template, nous lions le formulaire avec `[formGroup]="newsletterForm"`.
6. L'input utilise `formControlName="email"` pour le lier au contrôle du formulaire.
7. Le bouton de soumission est désactivé si le formulaire n'est pas valide.

Par ailleurs, `ReactiveFormsModule` joue un rôle crucial dans la mise en place des formulaires réactifs, il :

1. Fournit les directives essentielles pour les formulaires réactifs, comme `formGroup`, `formControlName`, et `formControl`. Ces directives sont utilisées dans le template HTML pour lier les éléments du formulaire à `FormGroup` et `FormControl` définis dans le composant.
2. Permet l'utilisation de `FormBuilder` : Il rend disponible le service `FormBuilder` que l'on injecte dans le

constructeur. `FormBuilder` facilite la création de formulaires réactifs.

3. Active la fonctionnalité des formulaires réactifs : En important ce module, vous "activez" essentiellement toute la fonctionnalité des formulaires réactifs dans votre composant.
4. Gère les mises à jour et la validation : Il fournit l'infrastructure nécessaire pour gérer les mises à jour d'état du formulaire et exécuter les validations que vous avez définies.
5. Permet la liaison de données bidirectionnelle : Il permet la synchronisation entre le modèle de formulaire dans votre composant et l'interface utilisateur dans le `template`.

Cet exemple montre comment créer un formulaire réactif basique, le lier au template, et gérer la soumission. Les Reactive Forms permettent une gestion plus robuste des formulaires, particulièrement utile pour des scénarios complexes avec de multiples champs et des validations personnalisées.

14.4 Service partagé

Il s'agit maintenant de partager l'information récupérée par le formulaire au composant d'affichage des tâches.

On peut pour cela utiliser un "service partagé" qui émettra la valeur du formulaire à chaque changement et il suffit de faire "souscrire" à cet observable tous les composants qui en ont besoin .

14.4.1 Ajout de loadTasks au service existant

Le service data-task.service.ts devient :

```
import Task from './shared/task.interface';
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class DataTasksService {
  constructor(private http: HttpClient) {}
  loadTasks(): Observable<Task[]> {
    const url = 'http://localhost:3000/tasks';
    const params = { status: 'PENDING' };
    return this.http.get<Array<Task>>(url, {
params });
  }
  private formValues = new BehaviorSubject<any>
```

```
(null);

  setFormValues(values: any) {
    this.formValues.next(values);
  }
  getFormValues() {
    return this.formValues.asObservable();
  }

}
```

Vous noterez BehaviorSubject qui est un type de sujet qui conserve la dernière valeur émise et l'émet immédiatement à tous les nouveaux abonnés.

Vous noterez également l'emploi de la méthode "asObservable" qui crée et renvoie un nouvel observable avec le sujet comme source.

14.4.2 Injection du service dans le composant formulaire

Le composant form-task.component.ts récupère la référence au service via une injection de service et appelle la méthode du service setFormValue qui va émettre une valeur (celle de l'input) via la méthode next():

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { DataTasksService } from '../data-
tasks.service';

@Component({
  selector: 'digi-form-task',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './form-task.component.html',
  styleUrl: './form-task.component.css',
})
export class FormTaskComponent {
  constructor(private dataTasksService:
DataTasksService) { }
  register(taskTitle: string) {
    console.log(`Formulaire soumis`, taskTitle);
    // Appel de la méthode du service qui émet la
valeur (next)

this.dataTasksService.setFormValues(taskTitle);

  }

}
```

14.4.3 Souscription du service dans le composant tasks-list.component.ts

Le composant form-task.component.ts récupère la référence au service via une injection de service et appelle la méthode du service setFormValue qui va émettre une valeur (celle de l'input) via la méthode next():

```
import { Component, Input } from '@angular/core';
import { CommonModule } from '@angular/common';
import Task from '../shared/task.interface';
import { TaskComponent } from
'../task/task.component';
import { DataTasksService } from '../data-
tasks.service';
import { FormTaskComponent } from '../form-
task/form-task.component';

@Component({
  selector: 'digi-tasks-list',
  standalone: true,
  imports: [CommonModule, TaskComponent,
FormTaskComponent],
  templateUrl: './tasks-list.component.html',
  styleUrl: './tasks-list.component.css',
})

export class TasksListComponent {
```

```
  tasks: Task[] = [];
  constructor(private dataTasksService:
DataTasksService) {}

  ngOnInit(): void {

this.dataTasksService.loadTasks().subscribe((tasks
: Task[]) => {
      this.tasks = tasks;
    });
    // souscription au service

this.dataTasksService.getFormValues().subscribe((v
alues) => {
      if (values) {
        console.log('Form values dans tasks-
list.component.ts :', values);
        this.tasks.push({ name: values.taskName,
id: 1000, done: false });
      }
    });
  }
  @Input() data: string = '';
  handleClickButtonValidate(taskId: number): void
{
    console.log(`Dans handleClickButtonValidate`);
    this.tasks = this.tasks
      .map((task) => {
        if (task.id == taskId) return { ...task,
```

```
done: !task.done };
        return task;
      })
      .sort((a: Task, b: Task) => Number(a.done) -
Number(b.done));
  }
}
```

Notons que dans cet exemple, toutes les nouvelles tâches auront le même id, ce qui n'est pas souhaitable. Plusieurs solutions sont possibles (Math.random(), Date.now().toString()) mais la plus sûre est l'utilisation du package uuid.
Pour notre exercice, nous nous contenterons d'utiliser Date.now() :

```
const newTask = { name: values.taskName, id:
Date.now(), done: false }
this.tasks.push(newTask);
```

14.4.4 Ajout de addTask au service existant

Nous avons besoin, afin de faire persister les données, d'exécuter une requête http avec le verbe "POST"
Voici ce que devient le fichier data-tasks.service.ts :

```
import Task from './shared/task.interface';
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, BehaviorSubject, catchError,
throwError } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class DataTasksService {
  static url = 'http://localhost:3000/tasks';
  constructor(private http: HttpClient) {}
  addTask(task: Task): Observable<any> {
    console.log(`Dans addTask de
DataTasksService`, task);
    return this.http.post<Task>
(DataTasksService.url, task);
  }

  loadTasks(): Observable<Task[]> {
    const params = { status: 'PENDING' };
    return this.http.get<Array<Task>>
(DataTasksService.url, { params });
  }

  private formValues = new BehaviorSubject<any>
(null);

  setFormValues(values: any) {
```

```
    this.formValues.next(values);
  }

  getFormValues() {
    return this.formValues.asObservable();
  }
}
```

Attention à souscrire au nouvel observable dans tasks-list.component.ts dans ngOnInit :

```
ngOnInit(): void {

this.dataTasksService.loadTasks().subscribe((tasks
: Task[]) => {
      this.tasks = tasks;
    });


this.dataTasksService.getFormValues().subscribe((v
alues) => {
      if (values) {
        console.log('Form values dans tasks-
list.component.ts:', values);
        const newTask = { name: values.taskName,
id: Date.now(), done: false };
        this.tasks.push(newTask);

this.dataTasksService.addTask(newTask).subscribe({
```

```
            next: (data) => {
                console.log(`data récupérée en retour
de addTask :`, data);
            },
            error: (error) => {
                console.error('Erreur attrapée :',
error);
            }}
          );
      }
    });
  }
```

14.5 Exercices

Exo

Au click sur le bouton "Valider/Invalider", faites en sorte que le fichier db.json soit mis à jour.
Vous ajouterez pour cela une méthode updateTask au service data-tasks.service.ts

Exo

Ajouter un bouton "Supprimer" à côté du bouton "Valider/Invalider"
Faire en sorte qu'au click sur ce bouton, la tâche

disparaisse (en local et sur db.json)
Il vous faudra (entre autres) ajouter une méthode deleteTask au service data-tasks.service.ts

Exo

Gestion des erreurs. S'il advenait une erreur lors de l'appel de loadTasks ou addTask ou deleteTask , faîte en sorte :

- qu'un message apparaissent sur l'interface de l'application
- que les tâches locales soient à nouveau synchronisées avec les tâches présentes en base de donnée

15 Service Workers

Référence :

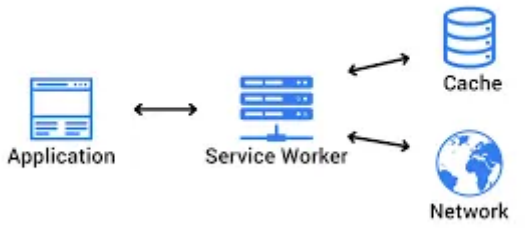
- https://v17.angular.io/guide/service-worker-intro
- https://v17.angular.io/guide/service-worker-getting-started

15.1 Introduction

L'ajout d'un service worker à une application Angular est l'une des étapes permettant de transformer une application en une application Web progressive (également appelée PWA).

Dans sa forme la plus simple, un service worker est un script qui s'exécute dans le navigateur Web et gère la mise en cache d'une application.

Les services workers fonctionnent comme un proxy réseau. Ils interceptent toutes les requêtes HTTP sortantes effectuées par l'application et peuvent choisir comment y répondre.



Par exemple, ils peuvent interroger un cache local et fournir une réponse en cache si elle est disponible. Le proxy ne se limite pas aux requêtes effectuées via des API programmatiques, telles que `fetch` ; il inclut également les ressources référencées dans `HTML` et même la requête initiale vers index.html.

Contrairement aux autres scripts qui composent une application, le service workers est conservé après la fermeture de l'onglet par l'utilisateur. La prochaine fois que le

navigateur charge l'application, le service workers se charge en premier et peut intercepter chaque demande de ressources pour charger l'application. Si le service workers est conçu pour le faire, il peut entièrement satisfaire le chargement de l'application, sans avoir besoin du réseau.

Même sur un réseau rapide et fiable, les retards aller-retour peuvent introduire une latence importante lors du chargement de l'application. L'utilisation d'un service workers pour réduire la dépendance au réseau peut améliorer considérablement l'expérience utilisateur.

15.2 ngsw-config.json

Pour prendre en charge ces comportements, le service workers Angular charge un fichier manifeste à partir du serveur. Le fichier, appelé `ngsw.json` décrit les ressources à mettre en cache et inclut les hachages du contenu de chaque fichier. Lorsqu'une mise à jour de l'application est déployée, le contenu du manifeste change, informant le service worker qu'une nouvelle version de l'application doit être téléchargée et mise en cache. Ce manifeste est généré à partir d'un fichier de configuration généré par la CLI appelé ngsw-config.json.

L'installation du service workers Angular est aussi simple que l'exécution d'une commande Angular CLI.

15.3 Tester le navigateur

Tous les navigateurs ne sont pas compatibles avec les services workers. cf https://caniuse.com/serviceworkers

Une façon de tester est d'exécuter le code suivant dans la console de votre navigateur :

```
if ('serviceWorker' in navigator) {
console.log('Service Worker is supported'); } else
{ console.log('Service Worker is not supported');
}
```

15.4 Installation

```
ng add @angular/pwa
```

cela :

- Ajoute le package @angular/service-worker à votre projet.
- Active la prise en charge de la création de service worker dans la CLI.
- Importe et enregistre le service worker auprès des fournisseurs racine de l'application.
- Met à jour le fichier index.html :
  Inclut un lien pour ajouter le fichier

manifest.webmanifest
Ajoute une balise méta pour theme-color

- Installe les fichiers d'icônes pour prendre en charge l'application Web progressive (PWA) installée.
- Crée le fichier de configuration du service worker appelé `ngsw-config.json`, qui spécifie les comportements de mise en cache et d'autres paramètres.

> ⚠ **Warning**
>
> Attention, ng serve ne fonctionne pas avec service Workers, il faudra utiliser http-server que vous pouvez installer
> `npm i -g http-server`

15.5 Exemple 1

Création d'une nouvelle application

```
ng new service-worker-demo
cd service-worker-demo
ng add @angular/pwa
```

La dernière commande ajout les fichier `ngsw-config.json`, `manifest.webmanifest` et des `icônes` (cf

`src/assets/icons`).

Notez :

Dans le fichier index.html, la ligne :

```
<link rel="manifest" href="manifest.webmanifest">
```

- Déclaration du manifeste : Elle indique au navigateur qu'il existe un fichier '"manifest.webmanifest") de manifeste pour cette application web. Le manifeste est un fichier JSON qui contient des métadonnées essentielles concernant votre PWA.
- Informations sur l'application : Le manifeste contient des informations cruciales telles que le nom de l'app, l'icône, les couleurs du thème, l'URL de démarrage, etc. Ces informations sont utilisées lorsque l'utilisateur installe l'app sur son appareil.
- Cette ligne, combinée au contenu du manifeste, permet au navigateur de reconnaître votre application comme "installable", ce qui active l'option "Ajouter à l'écran d'accueil" sur les appareils mobiles.
- Expérience hors ligne : En conjonction avec un Service Worker (également configuré par @angular/pwa), le manifeste aide à définir comment votre application doit se comporter hors ligne.

La création du fichier `ngsw-config.json`
Ce fichier est la configuration du Service Worker pour l'application PWA . En résumé, il :

- définit quels fichiers et ressources doivent être mis en cache par le Service Worker.
- spécifie comment ces ressources doivent être gérées et mises à jour.

15.5.1 build

```
ng build
```

Cette commande va générer l'application utilisable en production dans le fichier dist/service-worker-demo

15.5.2 Servir l'application

```
npx http-server -p 8080 -c-1 dist/service-worker-demo/browser
```

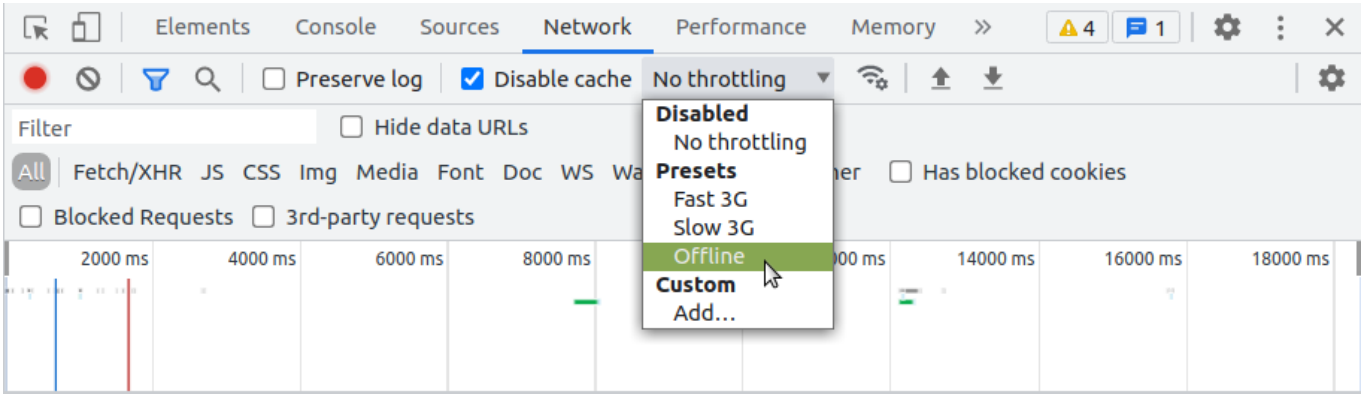
Cette commande va servir votre application avec le service worker à l'adresse http://localhost:8080.

> **Tip**

Lors du test des services workers Angular, c'est une bonne idée d'utiliser une fenêtre privée ou incognito dans votre navigateur pour garantir que le service worker ne lise pas un état précédent restant, ce qui peut provoquer un comportement inattendu.

15.5.3 Simuler un problème réseau

Dans votre navigateur (en mode privé) : Tools > Developer tool (F12) > Network tag > Select Offline in the Throttling dropdown menu :



L'application n'a désormais plus accès à l'interaction réseau.

Pour les applications qui n'utilisent pas le service worker Angular, l'actualisation afficherait la page de déconnexion Internet de Chrome indiquant « Il n'y a pas de connexion Internet ».

Avec l'ajout d'un service worker Angular, le comportement de l'application change. Lors d'une actualisation, la page se charge normalement.

Consultez l'onglet Réseau pour vérifier que le service worker est actif. Vous verrez que sous la colonne "Size", l'état des requêtes est (ServiceWorker). Cela signifie que les ressources ne sont pas chargées à partir du réseau. Au lieu de cela, elles sont chargées à partir du cache des Service Workers.

15.5.3.1 Qu'est-ce qui est mis en cache ?

Notez que tous les fichiers dont le navigateur a besoin pour restituer cette application sont mis en cache. La configuration standard ngsw-config.json est configurée pour mettre en cache les ressources spécifiques utilisées par la CLI :

index.html

- favicon.ico
- Artefacts de build (bundles JS et CSS)
- Tout ce qui se trouve sous assets
- Images et polices directement sous le outputPath configuré (par défaut ./dist/<project-name>/) ou resourcesOutputPath. Voir ng build pour plus d'informations sur ces options.
  Faites attention à deux points clés :

⚠ Warning

Le fichier ngsw-config.json généré inclut une liste limitée d'extensions de polices et d'images pouvant être mises en cache. Dans certains cas, vous souhaiterez peut-être modifier le modèle global en fonction de vos besoins.

Si les chemins d'accès aux ressourcesOutputPath ou aux ressources sont modifiés après la génération du fichier de configuration, vous devez modifier les chemins manuellement dans ngsw-config.json.

15.5.4 Apporter des modifications à votre application

Maintenant que vous avez vu comment les service workers mettent en cache votre application, l'étape suivante consiste à comprendre comment fonctionnent les mises à jour.

Apportez une modification à l'application et regardez le service worker installer cette mise à jour :

- Si vous effectuez un test dans une fenêtre de navigation privée, ouvrez un deuxième onglet vide. Cela permet de maintenir l'état de navigation privée et de cache actif pendant votre test.
- Fermez l'onglet de l'application, mais pas la fenêtre. Cela devrait également fermer les outils de développement.

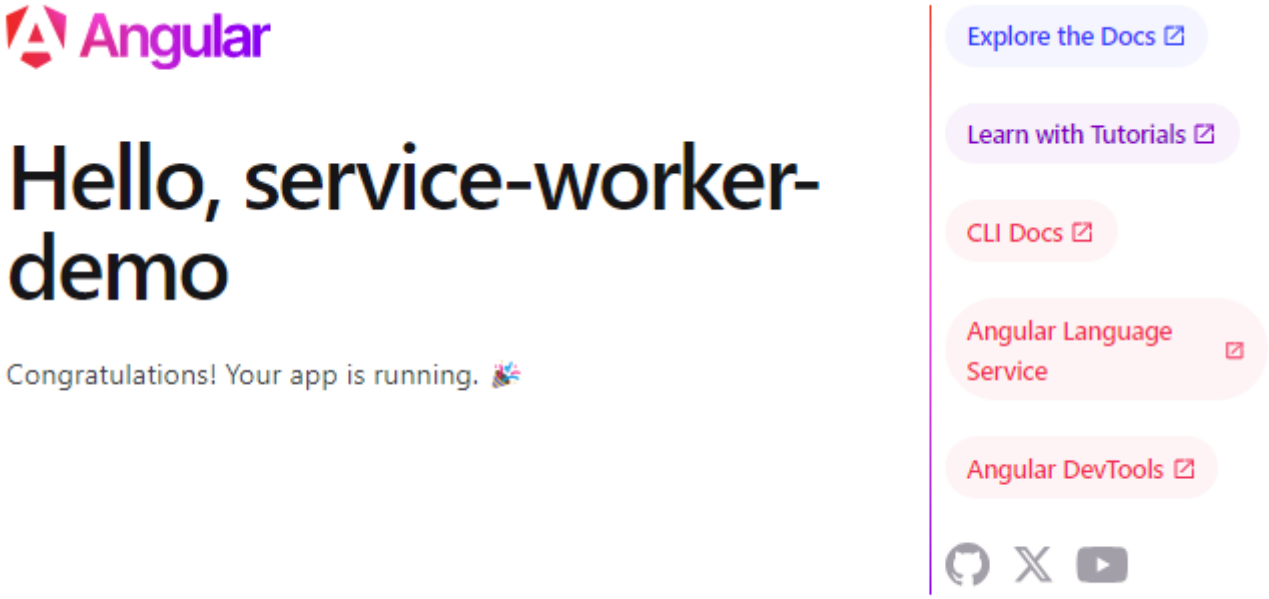
- Arrêtez le serveur http (Ctrl-c).
- Ouvrez `src/app/app.component.html` pour l'éditer.
- Remplacez le texte Welcome to {{title}}! par Bienvenue à {{title}}!.

Créez et exécutez à nouveau le serveur :

```
ng build
npx http-server -p 8080 -c-1 dist/service-worker-demo/browser
```

Voyons maintenant comment le navigateur et le service worker gèrent l'application mise à jour.

15.5.4.1 Ouvrez à nouveau http://localhost:8080 dans la même fenêtre. Que se passe-t-il ?



Rien, en fait !

Le service worker Angular fait son travail et sert la version de l'application qu'il a installée, même si une mise à jour est disponible. Dans un souci de rapidité, le service worker n'attend pas de vérifier les mises à jour avant de servir l'application qu'il a mise en cache.

Consultez les logs du serveur http pour voir le service worker demander /ngsw.json :
[2024-08-29T09:56:30.421Z] "GET /ngsw.json?ngsw-cache-bust=0.1838650352882165 2" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/128.0.0.0 Safari/537.36"

15.5.4.2 Rafraichir la page

Le service worker a installé la version mise à jour de votre application en arrière-plan, et la prochaine fois que la page est chargée ou rechargée, le service worker passe à la dernière version.

15.6 Exemple2

Téléchargez le fichier zip suivant et dézippez le. Cela devrait créer le répertoire `angularCrudProduct-main` et placez vous dessus via un terminal.

Si vous regardez le fichier README et que vous lancer json-server et l'application, vous verrez que cette application fait appel à une api REST pour sauvegarder les produits.

Le but de l'exercice est de faire en sorte que l'application fonctionne (à peu près) même si le navigateur fonctionne

hors ligne.

15.6.1 Installation de pwa

```
ng add @angular/pwa   # setup to use service worker
```

15.6.2 Gestion du cache

La gestions du cache peut se faire en partie via le fichier ngsw-config.json :

```
{
  "$schema": "./node_modules/@angular/service-worker/config/schema.json",
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.html",
          "/manifest.webmanifest",
          "/*.css",
          "/*.js"
        ]
```

```
      }
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "resources": {
        "files": [
          "/assets/**",
          "/media/*.
(svg|cur|jpg|jpeg|png|apng|webp|avif|gif|otf|ttf|w
off|woff2)"
        ]
      }
    }
  ],
  "dataGroups": [
    {
      "name": "products-api",
      "urls": ["http://localhost:3000/products"],
      "cacheConfig": {
        "strategy": "performance",
        "maxSize": 100,
        "maxAge": "3d",
        "timeout": "10s"
      }
    }
  ]
}
```

Ce fichier aura pour effet que la requête http://localhost:3000/products sera mise en cache pendant 3 jours, et le Service Worker essaiera de la mettre à jour en arrière-plan à chaque fois qu'elle sera demandée. Si la mise à jour prend plus de 10 secondes, il utilisera la version en cache.

Ce comportement vise à offrir des performances optimales en servant rapidement le contenu depuis le cache tout en le maintenant à jour en arrière-plan.

15.6.3 Ajout du service `Offlinequeue`

Créez le fichier src/app/offlinequeue.service.ts :

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, of, from } from 'rxjs';
import { catchError, tap, mergeMap } from
'rxjs/operators';

@Injectable({
  providedIn: 'root',
})

export class Offlinequeue {
  private readonly QUEUE_KEY = 'offline_queue';
  private isOnline: boolean = navigator.onLine;
```

```
  constructor(private http: HttpClient) {
    window.addEventListener('online', () =>
this.processQueue());
    window.addEventListener('offline', () =>
(this.isOnline = false));
  }

  addToQueue(url: string, method: string, body:
any): void {
    console.log(`Dans addToQueue`);
    const queue = this.getQueue();
    queue.push({ url, method, body });
    this.saveQueue(queue);
    console.log(`Etat de la queue :`, queue);
  }


  processQueue(): void {
    if (!this.isOnline) return;
    console.log(`Dans processQueue`);
    const queue = this.getQueue();
    if (queue.length === 0) return;
    from(queue)
      .pipe(
        mergeMap((item) =>
          this.sendRequest(item).pipe(
            catchError((error) => {
              console.error('Error processing
```

```
queued item:', error);
              return of(null);
            })
          )
        )
      )
      .subscribe(() => {
        this.clearQueue();
        this.isOnline = true;
      });
  }
  private sendRequest(item: any): Observable<any>
{
    return this.http.request(item.method,
item.url, { body: item.body }).pipe(
      tap(() => {
        const queue = this.getQueue();
        const index = queue.findIndex(
          (q) => q.url === item.url && q.method
=== item.method
        );
        if (index > -1) {
          queue.splice(index, 1);
          this.saveQueue(queue);
        }
      })
    );
  }
```

```
  private getQueue(): any[] {
    const queue =
localStorage.getItem(this.QUEUE_KEY);
    return queue ? JSON.parse(queue) : [];
  }

  private saveQueue(queue: any[]): void {
    localStorage.setItem(this.QUEUE_KEY,
JSON.stringify(queue));
  }

  private clearQueue(): void {
    localStorage.removeItem(this.QUEUE_KEY);
  }
}
```

15.6.4 Appel du service `Offlinequeue`

Modification de products.service.ts :

```
import { Injectable } from '@angular/core';
import {
  NewProductInterface,
  PatchProductInterface,
  ProductInterface,
} from './interfaces/product.interface';
import { HttpClient, HttpErrorResponse } from
'@angular/common/http';
import { Observable, BehaviorSubject, catchError,
```

```
throwError, of } from 'rxjs';

import { Offlinequeue } from
'./offlinequeue.service';

@Injectable({
  providedIn: 'root',
})

export class ProductsService {
  static url = 'http://localhost:3000/products';

  constructor(private http: HttpClient, private
offlineQueue: Offlinequeue) {}

  addProduct(
    product: NewProductInterface
  ): Observable<ProductInterface | null> {
    console.log(`Dans addProduct de
ProductsService`, product);

    if (navigator.onLine) {
      console.log(`Navigateur en ligne dans
addProduct`);
      this.offlineQueue.processQueue();
      return this.http
        .post<ProductInterface>
(ProductsService.url, product)
        .pipe(
```

```
          catchError((error: HttpErrorResponse) =>
{
            console.error(`Erreur lors de l'ajout
du produit:`, error);
            return of(null);
          })
        );
    } else {
      console.log(`Navigateur hors ligne`);
      // Ajouter à la file d'attente hors ligne

this.offlineQueue.addToQueue(ProductsService.url,
'POST', product);
      // Retourner un Observable avec un produit
simulé (sans id)
      return of({
        ...product,
        id: 'Tmp-' + Date.now().toString(),
      } as ProductInterface);
    }
  }

  loadProducts(): Observable<ProductInterface[]> {
    if (navigator.onLine) {
      console.log(
        `Navigateur en ligne dans loadProducts :
appel de processQueue`
      );
      this.offlineQueue.processQueue();
```

```
    }
    console.log(`Dans loadProduct de
ProductsService`);
    return this.http.get<ProductInterface[]>
(ProductsService.url);
  }

  loadOneProduct(id: string):
Observable<ProductInterface> {
    console.log(`Dans loadOneProduct de
ProductsService`);

    return this.http.get<ProductInterface>
(`${ProductsService.url}/${id}`);
  }



  patchProduct(
    id: string,
    partialProduct: PatchProductInterface
  ): Observable<ProductInterface> {
    console.log(`Dans patchProduct de
ProductsService`);
    return this.http.patch<ProductInterface>(
      `${ProductsService.url}/${id}`,
      partialProduct
    );
  }
  deleteProduct(id: string):
```

```
Observable<ProductInterface> {
    console.log(`Dans deleteProduct de
ProductsService`);

    return this.http.delete<ProductInterface>
(`${ProductsService.url}/${id}`);
  }
}
```

Vous noterez que lorsque le navigateur est hors ligne, addProduct :

- Ajoute à la 'queue' le produit qui vient d'être posté
- renvoie un produit avec un id qui commence par 'Tmp'

Lorsque le navigateur est en ligne, on exécute la queue avec l'instruction :

```
this.offlineQueue.processQueue();
```

15.6.5 Ajout d'une nouvelle route

Fichier app.routes.

```
import { Routes } from '@angular/router';

import { ProductAddComponent } from './product-
add/product-add.component';
```

```
import { ProductEditComponent } from './product-edit/product-edit.component';

import { ProductGetComponent } from './product-get/product-get.component';

import { NewsletterComponent } from './newsletter/newsletter.component';

import { ProductTmpComponent } from './product-tmp/product-tmp.component';



export const routes: Routes = [

  { path: 'add', component: ProductAddComponent },

  { path: 'edit/:id', component: ProductEditComponent },

  { path: 'get', component: ProductGetComponent },

  { path: 'newsletter', component: NewsletterComponent },

  { path: 'tmp', component: ProductTmpComponent },

];
```

Ajout d'un nouveau composant

```
ng g c product-tmp
```

Modification du template product-tmp.component.html :

```
<h2>Vous venez d'ajouter un produit alors que vous
n'êtes pas en ligne. <br>L'ajout du

produit est en liste d'attente et se fera lorque
vous serez à nouveau en ligne</h2>
```

15.6.6 Modification de product-add.component.ts

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  FormBuilder,
  FormGroup,
  Validators,
  ReactiveFormsModule,
} from '@angular/forms';

import { NewProductInterface } from
'../interfaces/product.interface';
```

```
import { ProductsService } from
'../products.service';

import { Router } from '@angular/router';

@Component({
  selector: 'app-product-add',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './product-add.component.html',
  styleUrl: './product-add.component.css',
})

export class ProductAddComponent implements OnInit
{
  productForm!: FormGroup; //l'opérateur de non-
nullité (!) indique à TypeScript que l'on est
certain que

  // productForm sera initialisé avant d'être
utilisé
  isSubmitted = false;

  constructor(
    private fb: FormBuilder,
    private productsService: ProductsService,
    private router: Router
  ) {}
```

```
  ngOnInit() {
    this.productForm = this.fb.group({
      name: ['', Validators.required],
      description: [''],
      price: [null, [Validators.required,
Validators.min(0)]],
    });

  }

  register() {
    this.isSubmitted = true;
    if (this.productForm.valid) {
      const product: NewProductInterface =
this.productForm.value;
      console.log(`Ajout du produit`, product);
      // Souscription à l'observable retourné par
addProduct

this.productsService.addProduct(product).subscribe
({

        next: (response) => {

          console.log('Produit ajouté avec
succès:', response);

          // Réinitialisez le formulaire ou
naviguez vers une autre page
```

```
          this.productForm.reset();

          this.isSubmitted = false;

          if (response) {

            if (response.id.startsWith('Tmp'))
this.router.navigate(['/tmp']);

            else this.router.navigate(['/get']);
          }
        },
        error: (error) => {

          console.error(`Erreur lors de l'ajout du
produit:`, error);

          // Gérez l'erreur (par exemple, affichez
un message à l'utilisateur)
        },
      });

    } else console.log(`Problème lors de l'ajout
du formulaire`);
  }
}
```

Vous noterez qu'en fonction du retour de l'appel de `addProduct()`, l'utilisateur est redirigé vers une page différente :

```
if (response.id.startsWith('Tmp'))
this.router.navigate(['/tmp']);
else this.router.navigate(['/get']);
```

15.6.7 Lancer l'application

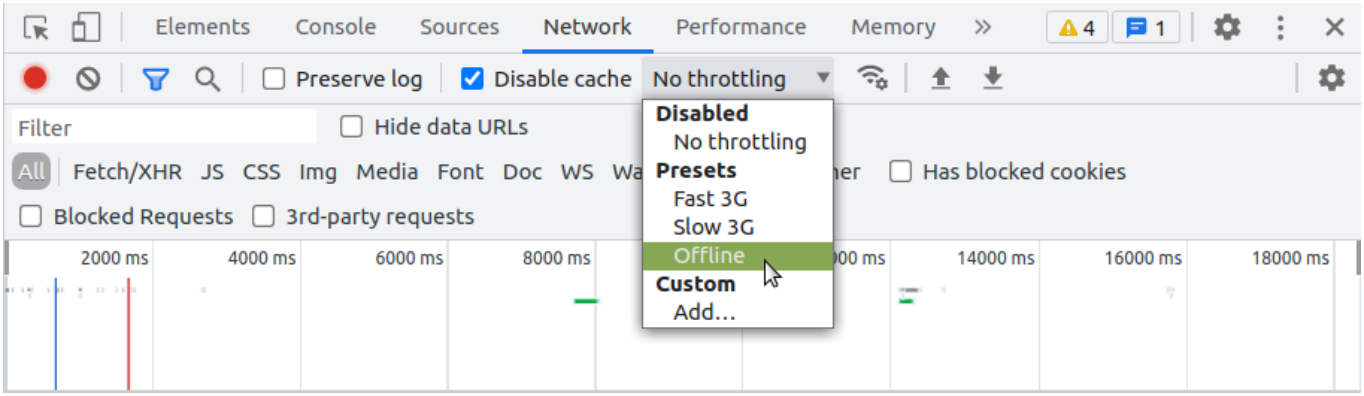
```
ng build
```

puis dans un autre terminal

```
npx http-server -p 8080 -c-1
dist/Angularcrud/browser
```

15.6.8 Simuler un problème réseau

Dans votre navigateur : Tools > Developer tool (F12) > Network tag > Select Offline in the Throttling dropdown menu :



L'application n'a désormais plus accès à l'interaction réseau.

Pour les applications qui n'utilisent pas le service worker Angular, l'actualisation afficherait la page de déconnexion Internet de Chrome indiquant « Il n'y a pas de connexion Internet ».

Si vous essayez d'ajouter un nouveau produit alors que vous êtes hors connexion, vous êtes renvoyée sur la route `tmp`

L'ajout du produit (avec l'appel à l'API REST) ne se fera donc que lorsque vous serez à nouveau en ligne via l'appel de la méthode `processQueue` du service `Offlinequeue`

16 Tests

Angular permet 2 types de tests :

- tests unitaires
- test end-to-end
  Les premiers sont là pour affirmer qu'une petite portion de code (un composant, un service, un pipe) fonctionne

correctement en isolation, c’est à dire indépendamment de ses dépendances. Écrire un tel test unitaire demande d’exécuter chacune des méthodes d’un composant/service/pipe, et de vérifier que les sorties sont celles attendues pour les entrées fournies. On peut aussi vérifier que les dépendances utilisées par cette portion sont correctement invoquées, par exemple qu’un service fait la bonne requête HTTP.

On peut aussi écrire des tests end-to-end ("de bout en bout"). Ceux-ci émulent une interaction utilisateur réelle avec l'application, en démarrant une vraie instance et en pilotant le navigateur pour saisir des valeurs dans les champs, cliquer sur les boutons, etc. On pourra par exemple vérifier que la page affichée contient ce qui est attendu ou que l’URL est correcte

16.1 Tests unitaires

Ces tests sont rapides et ils sont très efficaces pour tester les cas limites, qui peuvent être difficiles à reproduire manuellement dans l’application réelle.

L’un des concepts au cœur d’un test unitaire est celui d’isolation : on ne veut pas que notre test soit biaisé par ses dépendances. Alors on utilise généralement des objets

bouchonnés (mock) comme dépendances. Ce sont des objets factices créés juste pour les besoins du test.

La bibliothèque "Jasmine" offre plusieurs méthodes pour déclarer des tests :

- describe() déclare une suite de tests (un groupe de tests) ;
- it() déclare un test ;
- expect() déclare une assertion.

Un test JavaScript basique ressemble à :

```
class Pony {
  constructor(public name: string, public speed:
number) {}
  isFasterThan(speed: number): boolean {
  return this.speed > speed;
  }
}
describe('My first test suite', () => {
  it('should construct a Pony', () => {
  const pony = new Pony('Rainbow Dash', 10);
  expect(pony.name).toBe('Rainbow Dash');
  expect(pony.speed).not.toBe(1);
  expect(pony.isFasterThan(8)).toBe(true);
  });
});
```

L’appel à expect() peut être chaîné avec des méthodes comme toBe(), toBeLessThan(), toBeUndefined(), etc. Chaque méthode peut être rendue négative avec l’attribut not de l’objet retourné par expect().
Le fichier de test est un fichier séparé du code à tester, en général avec une extension comme .spec.ts.

Le test d’une classe Pony écrite dans un fichier pony.ts sera normalement dans un fichier nommé pony.spec.ts. On peut soit poser ce fichier de test juste à côté du fichier à tester, soit dans un répertoire dédié contenant tous les tests.

La méthode beforeEach() initialise un contexte avec chaque test. Si j’ai plusieurs tests sur le même poney, il est préférable d’utiliser beforeEach() pour initialiser ce poney, plutôt que copier/coller le même morceau de code dans tous les tests.

Par exemple :

```
describe('Pony', () => {
  let pony: Pony;
  beforeEach(() => {
  pony = new Pony('Rainbow Dash', 10);
  });
  it('should have a name', () => {
  expect(pony.name).toBe('Rainbow Dash');
  });
  it('should have a speed', () => {
```

```
    expect(pony.speed).not.toBe(1);
    expect(pony.speed).toBeGreaterThan(9);
    });
});
```

16.1.1 Bouchons

"Jasmine" nous permet de créer des objets factices (bouchons), ou même d'espionner une méthode d'un véritable objet. On peut alors faire quelques assertions sur ces méthodes, comme toHaveBeenCalled() qui vérifie que la méthode a bien été invoquée, ou toHaveBeenCalledWith() qui vérifie les paramètres exacts utilisés lors de l'invocation de la méthode espionnée.

Tu peux aussi vérifier combien de fois la méthode a été invoquée, ou vérifier si elle ne l'a pas été, etc. Par exemple, si l'on a une classe Race avec une méthode start(), qui appelle run() sur chaque poney dans la course, et filtre ceux qui n'ont pas commencé à courir (run() renvoie un booléen) :