

ANGULAR



ANGULAR

Framework web complet :

- Moteur de rendu/templating
 - 3 modes d'executions:
 - client side (CSR) => 100% coté navigateur
 - pré-rendering (SSG) => semi client side/ server side
 - server side (SSR) => 99% coté serveur
- CLI
 - compiler
 - bundler (webpack ou esbuild)
 - générateur de code
- Composants

ANGULAR

Quelques liens (très) utiles:

- doc angular: <https://angular.dev/>
 - /!\ vous pouvez parfois tomber sur angular.io qui est l'ancienne adresse, c'est légitime pour les versions < 17
- doc angular Material: <https://material.angular.io/>
 - Toujours en .io ...
- Repo Angular: <https://github.com/angular/angular>
- Listes des features introduites: <https://www.angular.courses/caniuse>

HISTORIQUE

- Angular a été créé par deux ingénieurs de Google (**Miško Hevery** et **Adam Abrons**) en 2009 avec AngularJS.

AngularJS => framework en javascript

Angular “2” => repart de zéro et devient le framework que l’on connaît aujourd’hui. (2016)

- il n’existe pas de version 3, afin d’aligner les différents packages (routeur déjà en v3)

■

1 version majeure tous les 6 mois: la version actuelle est 19 => c’est bien mais faut suivre...

LES VERSIONS

Actively supported versions

The following table provides the status for Angular versions under support.

Version	Status	Released	Active ends	LTS ends
^19.0.0	Active	2024-11-19	2025-05-19	2026-05-19
^18.0.0	LTS	2024-05-22	2024-11-19	2025-11-19
^17.0.0	LTS	2023-11-08	2024-05-08	2025-05-15

<https://angular.dev/reference/releases#release-schedule>

ANGULAR "RENAISSANCE"

Depuis l'année dernière (v16/v17):

- Simplification du framework
- Modernisation du framework
 - Modification structurante
 - une app v15 ne ressemble plus du tout à une app v19

Ce cours traite de la dernière version, mais aborde les anciens concepts encore existant tout de même

DÉMO

Un vrai projet angular

PRÉREQUIS

- Node: angular cli tourne avec node
- Typescript: langage de prog d'angular
- un peu de css (voir scss)
- Un IDE: vscode, intelij,...
 - Pour les utilisateurs de VSCode
 - installer l'extension **Angular Language Service**

ANGULAR CLI

ANGULAR CLI

Permet de gérer tout le cycle de développement de l'application :

- Génération d'un projet, d'un composant, d'un service,...
- Mise à jour des versions
- Migration de code entre les versions
- Tests (unitaires)
- Compilation :
 - pour le développement
 - pour la production

ANGULAR CLI

Pour les tâches communes, Angular utilise en fait plusieurs outils du "marché", les plus remarquables :

- Compilation :
 - webpack
 - esbuild
 - typescript
 - ...
- Tests:
 - jasmine
 - karma,
 - (jest en beta, viendra bientôt vite aussi)

ANGULAR CLI

Il s'utilise de la façon suivante :

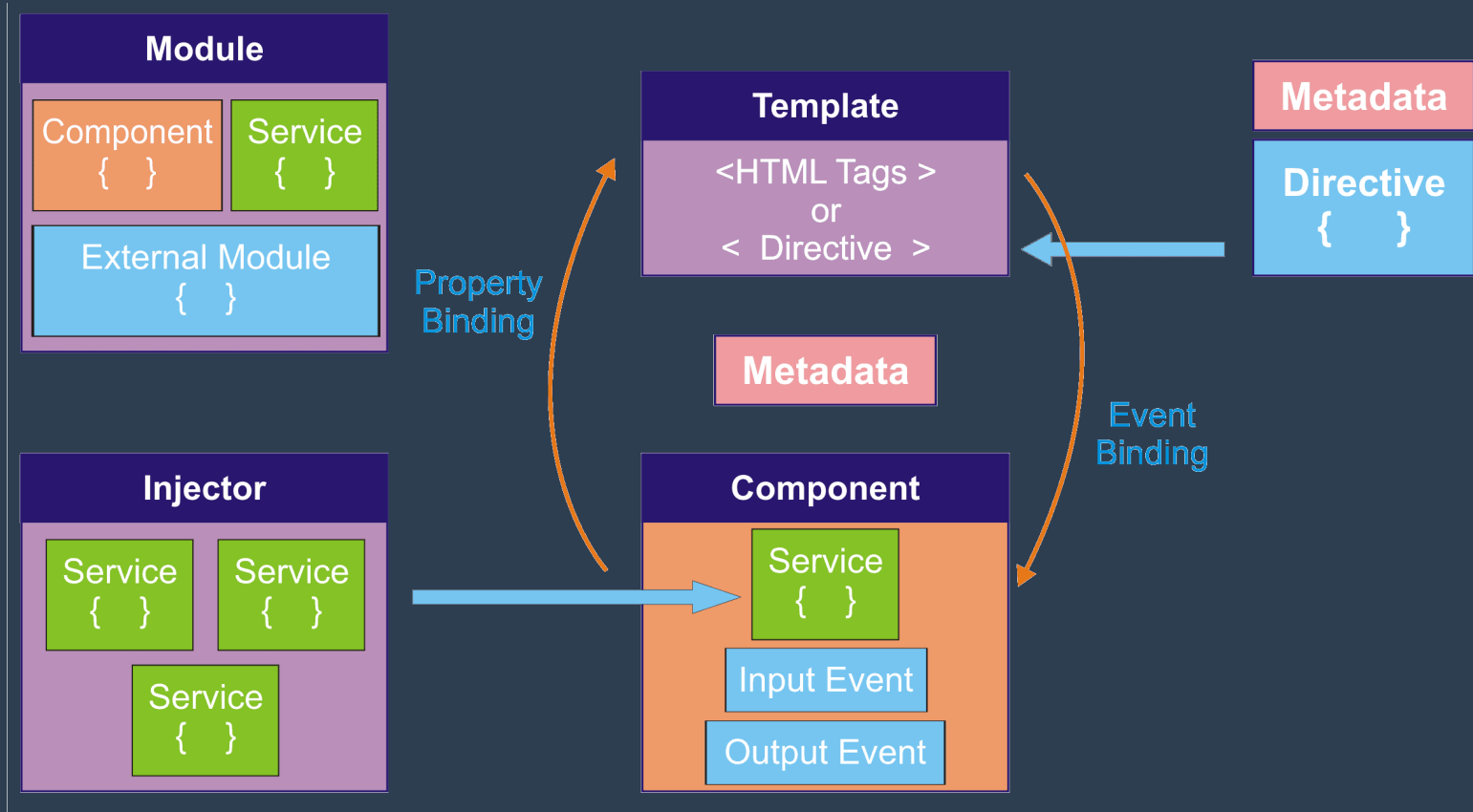
```
1  ng <commande> <option>
2
3  ng new mon-app
4
5  ng generate composant User
6  ng g c User
```

START CODING

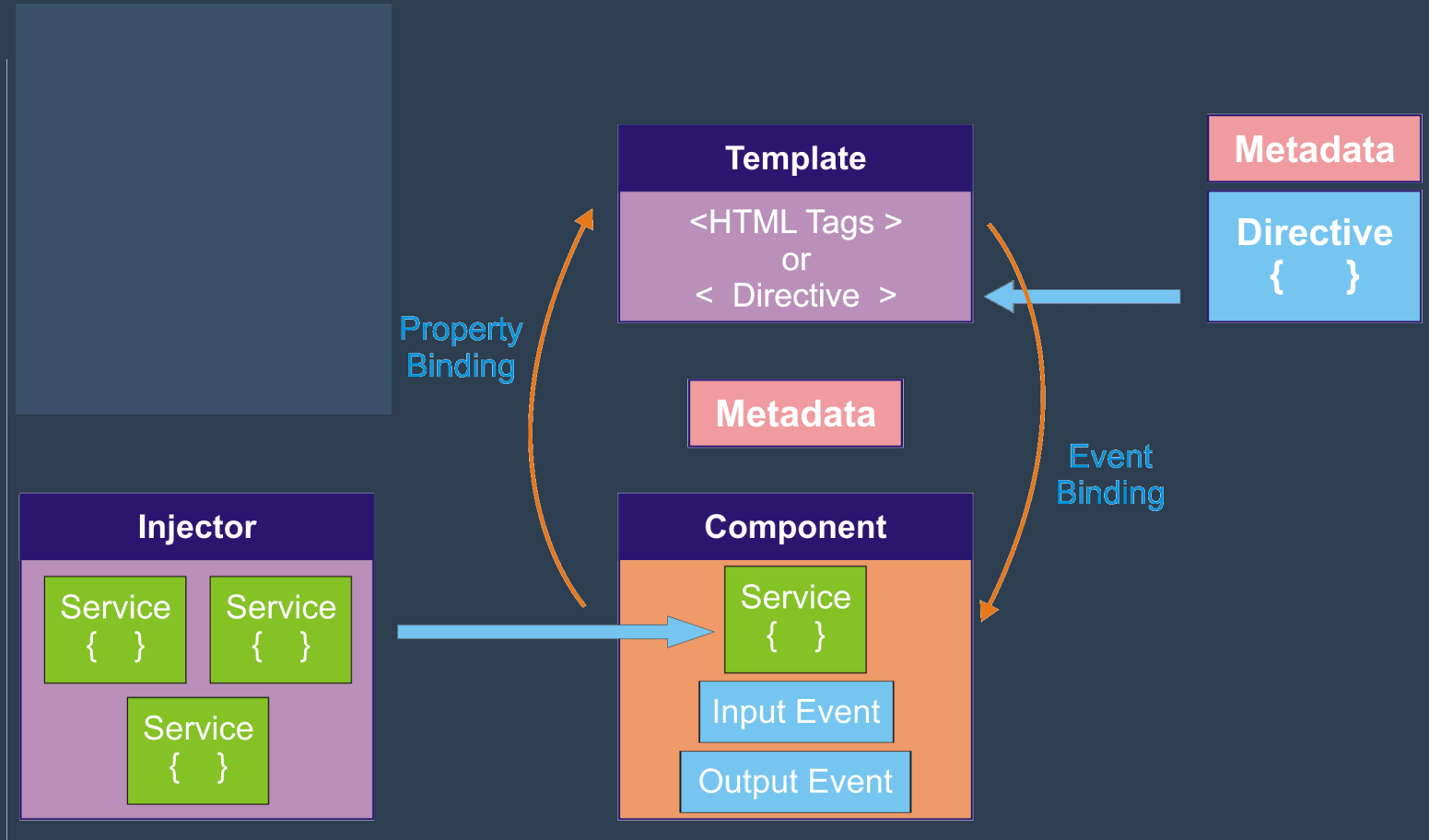
```
1 # Installer Angular cli
2 npm install -g @angular/cli
3
4 # Créer un nouveau projet
5 ng new mon-projet
6
7 # Lancer l'application
8 cd mon-projet
9 ng serve
10
11 # Ouvrir l'app avec l'url suivante :
12 http://localhost:4200
```

ARCHITECTURE

ARCHITECTURE - LEGACY

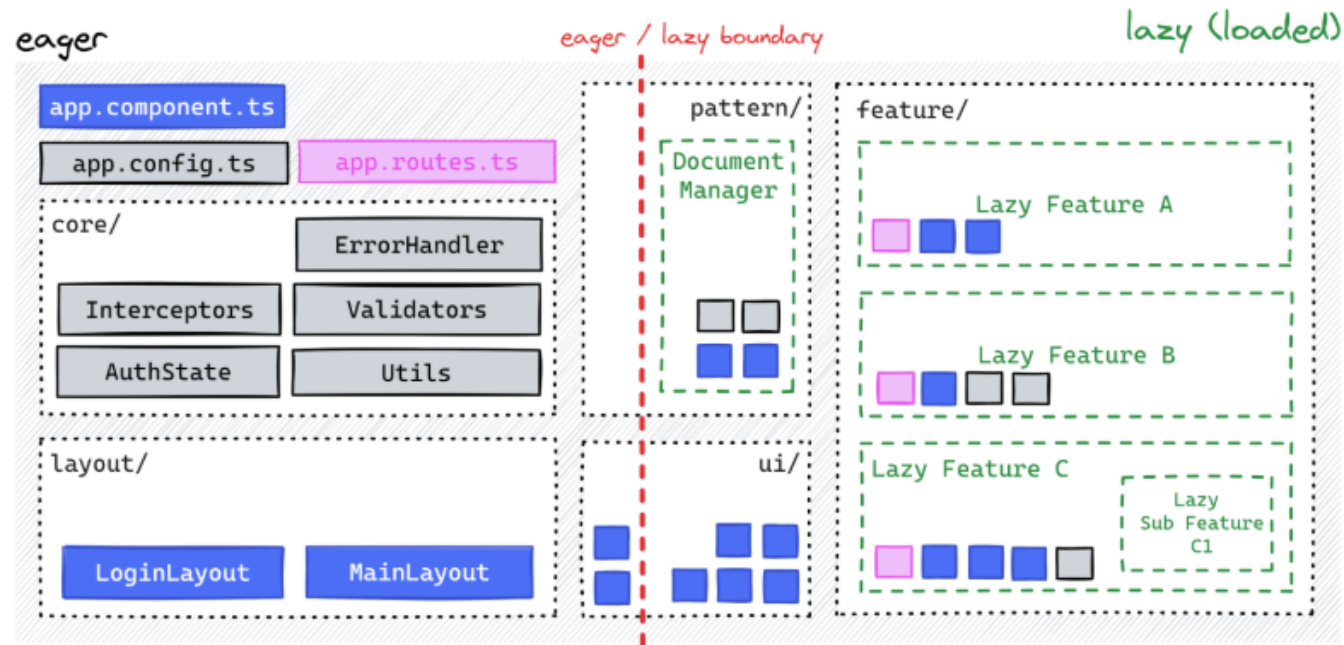


ARCHITECTURE - MODERNE



ARCHITECTURE

Universal Angular Application Architecture



⋮ Type folder

⋮ Pattern / Feature folder

■ Components / pipes / directives (declarables / template related / standalone)

■ Services / state management and other injectables (without template)

■ Routes (route config, providers config for lazy feature)

APPLICATION

Toutes applications a au moins un fichier main.ts et index.html

- Le main.ts va contenir les informations du haut de notre arbre de composants, services, ...
- l'index.html sera le point d'entrée de notre application sur le navigateur client

LES COMPOSANTS

LES COMPOSANTS

Un composant est construit **obligatoirement** de:


- une classe avec la logique le composant
 - accompagné d'un décorateur (équivalent de l'annotation java)
- un template :
 - situé dans un fichier .html
 - dans la déclaration du composant
- D'un nom "html": le selecteur
 - par convention: <abbreviation application>-<nom composant kebab case>

```
1 @Composant(...)
2 class AppComposant {}
```

LES COMPOSANTS

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6 })
7 export class AppComponent {}
```

On parle de
template inline



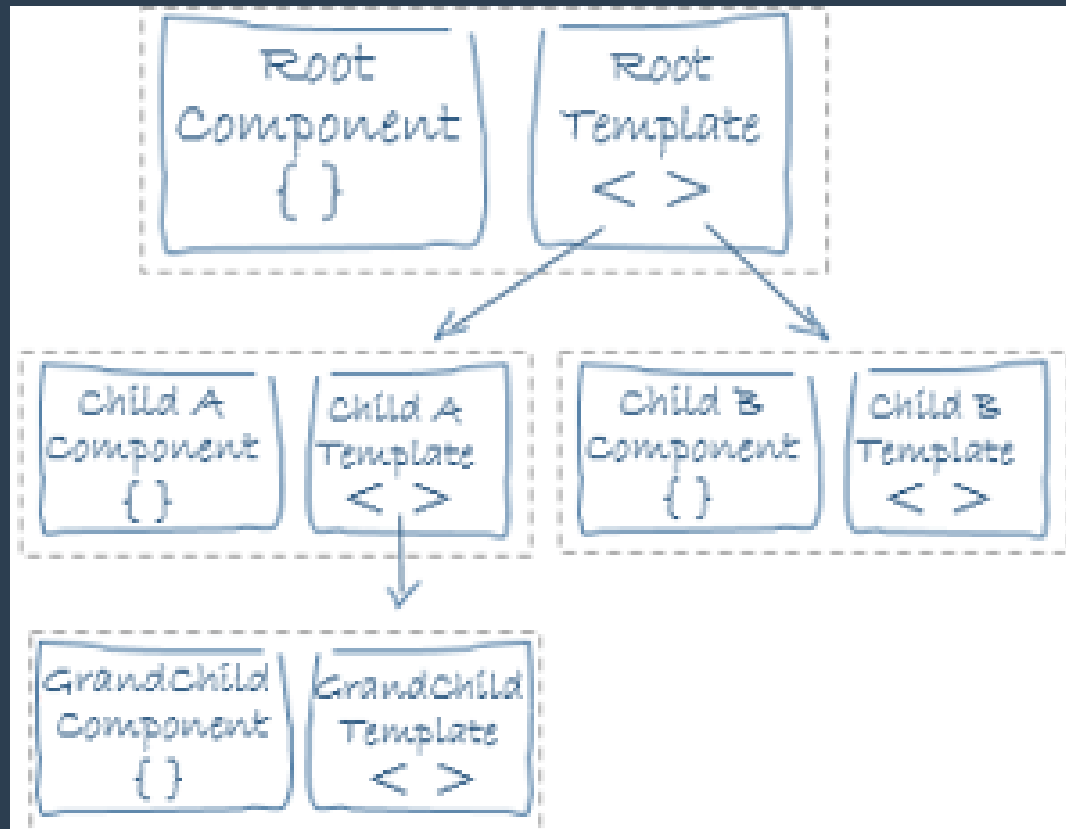
```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   template: `<h1>Hello</h1>`,
6 })
7 export class AppComponent {}
```

LES COMPOSANTS

Pour styliser notre composant, on va ajouter une métadate: `styleUrl` ou `styleUrls` ou `styles`.
(Un des trois)

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   styleUrl: './app.component.scss',
6   styleUrls: ['./app.component.scss'],
7   styles: `
8     h1 { color: red }
9   `,
10 })
11 export class AppComponent {}
```

LES COMPOSANTS Utiliser un composant



LES COMPOSANTS Utiliser un composant

```
1 import { Component } from '@angular/core';
2
3 @Component({
4     selector: 'app-hello-world',
5     template: `<h1>Hello world</h1>`
6 })
7 export class AppHelloWorld {}
8
9
10 @Component({
11     selector: 'app-root',
12     imports: [AppHelloWorld],
13     template: `
14         <app-hello-world />
15     `,
16 })
17 export class AppComponent {}
```


LES COMPOSANTS

Le template est "static", pour afficher le contenu d'une variable on va utiliser la syntaxe moustache: {{ maVariable }}

```
1 import { Component } from '@angular/core';
2
3 @Component({
4     selector: 'app-hello-world',
5     template: `<h1>Hello {{ nom }}, on est en {{ annee }}.</h1>`
6 })
7 export class AppHelloWorld {
8     nom = "Jean";
9     annee = 2025;
10 }
```

LES COMPOSANTS

à l'intérieur des moustaches on peut y faire des opérations simple

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-hello-world',
5   template: `
6     <h1>On est en {{ annee + 1 }}</h1>
7     <span>Hello {{ user.nom }} {{ user.prenom }}</span>
8   `
9 })
10 export class AppHelloWorld {
11   annee = 2024;
12   user = {
13     nom: "Bon",
14     prenom: "Jean"
15   }
16 }
```

LES COMPOSANTS

De la même manière que précédemment, on va pouvoir appeler une fonction dans le template:

/!\ Peut poser de lourd problème de performance !

=> au point où je l'interdit dans mes projets

=> avec la modernisation d'angular ça ne sera plus forcément un problème

```
1 import { Component } from '@angular/core';
2
3 @Component({
4     selector: 'app-hello-world',
5     template: `<h1>Hello {{ getNom() }}</h1>`
6 })
7 export class AppHelloWorld {
8     private nom = "Jean";
9
10    public getNom(): string {
11        return nom;
12    }
13 }
```

CYCLES DE VIE

- Permet d'effectuer des opérations selon l'état du composant, dans l'ordre d'exécution:
 - **Constructeur**
 - **ngOnInit**
 - ngOnChanges
 - ngDoCheck
 - ngAfterContentInit
 - ngAfterContentChecked
 - **ngAfterViewInit**
 - ngAfterViewChecked
 - **ngOnDestroy**

<https://angular.dev/guide/components/lifecycle#summary>

CYCLES DE VIE

- exemple avec ngOnInit

```
1  @Component({
2    selector: 'app-test',
3    template: ``
4  })
5  export class AppTest implements OnInit {
6    @Input() toto?: string;
7
8    constructeur() {
9      console.log(this.toto); // sera undefined
10   }
11
12   ngOnInit() {
13     console.log(this.toto); // affichera la bonne valeur !
14   }
15 }
```

LET'S GO

- Créer un composant qui affiche votre nom et prénom (en utilisant des propriétés de class)
- ajouter une feuille de style lié au composant pour afficher le prénom en couleur

Les signaux

COMMENT FONCTIONNE ANGULAR

Angular "patch" (monkey patching), grâce à la librairie zone.js, toutes les api de type évènement du navigateur :
click, setTimeout, appel http, ...

C'est grâce à ça que la magie d'Angular opère. Le changement sont ainsi automatiquement détecté et la vue rafraichi.

Zone permet aussi d'intercepter les erreurs du navigateur

Problème: c'est lourd, oblige à reconstruire tout l'arbre de composant, => performance impactées

COMMENT FONCTIONNE ANGULAR

Pour résoudre ça Angular à introduit les signaux (api Signal)

Les signaux vont permettre de suivre une modification de propriété.

Avec ça on va pouvoir tracer et mettre à jour plus finement l'arbre de composant.

Toute cette intégration est en cours coté framework et évolue au fil des versions. Aujourd'hui zone reste nécessaire. On peut le désactiver, mais c'est encore expérimentale

LES SIGNAUX

Créer un signal :

```
1 import { signal } from '@angular/core';  
2  
3 const age = signal(28);
```

Mettre à jour : (attention, il faut retourner un nouvelle instance d'un objet)

```
1 import { signal } from '@angular/core';  
2  
3 const age.update(a => a + 1);
```

Changer une valeur

```
1 import { signal } from '@angular/core';  
2  
3 const age.set(29);
```

LES SIGNALS

Récupérer la valeur - au moment de l'exécution

signal() retourne une fonction, donc on l'appelle simplement

```
1 import { signal } from '@angular/core';
2
3 const age = signal(28);
4
5 console.log(age());
```

écouter les modifications:

Ainsi chaque mise à jour (update/set) on va exécuter le code dans

effect

```
1 import { signal } from '@angular/core';
2
3 const age = signal(28);
4
5 effect(() => {
6   console.log(age());
7 });
```

LES SIGNAUX

Effect est encore en *developer preview*, donc il faut limiter son utilisation. Pratique pour debug surtout.

L'écoute des signaux dans le template angular est automatiquement faite.

LES SIGNAUX

Creation d'un signal dérivé: computed

- il va se réexécuter à chaque maj des signaux utilisés
- retourne lui même un signal, mais en lecture seul (pas d'update manuelle)

```
1 import { signal } from '@angular/core';
2
3 const nom = signal("Bon");
4 const prenom = signal("Jean");
5
6 const nomComplet = computed(() => `${prenom()} ${nom()}`);
7
8 console.log(nomComplet());
9
10 // Jean Bon
```

Dans cette exemple, à chaque modification de nom ou de prenom, nomComplet va changer aussi

LES SIGNAUX

Dans ce contexte de modernisation du framework, ils existent souvent des api avec une version "signal" et "classique". Je le préciserai si c'est le cas.

L'utilisation de l'un ou l'autre va dépendre:

- une app qui utilise la version classique:
 - ça va dépendre de la volonté/budget de l'équipe
 - sachant qu'angular propose des outils de migration dans certain cas
- une nouvelle app :
 - je conseille de partir sur du signal à 100%, et surtout pas mixer les deux

LET'S GO

à l'intérieur du précédent composant, transformer nom et prénom en signal et ajouter l'age, créer un computed qui va permettre de récupérer le nom complet et l'age et l'utiliser dans le template.

Dans le constructeur, ajouter un setInterval qui toute les x secondes ajoute +1 à l'age => ça va mettre à jour tout seul dans le template.

DATA BINDING

INPUT

- Permet de fournir des paramètres des composants père à un composant fils.
- Coté parent on va passer une valeur de la façon suivante:
 - maPropriete est public ou protected dans le père

```
1 <app-hello-world [monInput]="maPropriete" />
```

- Coté fils on va récupérer de la façon suivante:

```
1 // Signal
2 public monInput: Signal<string | undefined> = input<string>
3
4 // Classique
5 @Input() public monInput?: string;
```

INPUT

- Pour rendre le paramètre obligatoire:

```
1 // Signal
2 public monInput: string = input.required<string>();
3
4 // Classique
5 @Input({ required: true }) public monInput!: string;
```

- Renommer le paramètre

```
1 // Signal
2 public monInput: string
3     = input.required<string>({alias: 'monSuperInput'});
4
5 // Classique
6 @Input({ required: true, alias: 'monSuperInput' })
7 public monInput!: string;
```

INPUT

- Si mon input est une string alors je peux écrire :

```
1 <app-test [title]='Mon gros titre' />
2
3 // devient
4
5 <app-test title="Mon gros titre" />
```

OUTPUT

- Permet de fournir des informations du fils vers le père
- les outputs ne passe pas un valeur directement, mais un évènement
- Coté parent on va passer une valeur de la façon suivante:

```
1 <app-hello-world (monOutput)="handleOutput($event)" />
```

- \$event représente la valeur envoyé par le fils
- handleOutput est une méthode du père

OUTPUT

Coté fils :

```
1 // Signal
2 public monOutput = output<string>();
3
4 // Classique
5 @Output() public monOutput: EventEmitter<string>
6     = new EventEmitter();
7
8 public onClick(): void {
9     //signal et classique: même appel !
10    this.monOutput.emit('toto');
```

TWO WAY BINDING

Quand on veut garder une propriété synchronisé entre père et fils, on va utiliser la syntaxe "banana in the box"

```
1 <app [(parametre)]="maPropriete" />
```

Du cote fils:

- en signal, on va utiliser `model ()`
- en classique, on aura un input et un output avec comme nom: "`<input>Change`"

TWO WAY BINDING

Quand on veut garder une propriété synchronisé entre père et fils, on va utiliser la syntaxe "banana in the box"

Du cote fils:

- en signal, on va utiliser `model ()`
- en classique, on aura un input et un output avec comme nom: "`<input>Change`"

```
1 <app [(parametre)]="maPropriete" />
2
3 //revient à écrire
4 <app
5     [(parametre)]="maPropriete"
6     (parametreChange)="maPropriete = $event"
7 />
```

TWO WAY BINDING

Angular fournir un directive (on verra plus tard) qui simplifie la connexion entre un composant et un élément de formulaire: `ngModel`

```
1 <input [(ngModel)]="nom" />
2
3
4 Ma valeur: {{ nom }}
```

Par ailleurs ce code pourrait être écrit aussi grâce aux variables locales, que l'on définit avec un `#`:

```
1 <input #nom />
2
3
4 Ma valeur: {{ nom.valeur }}
```


JS EVENT

Les événements natif sont utilisé de la même façon qu'un output :

```
1 <app (click)="onClick($event)" />
2 <app (focus)="onlick($event)" />
3 <app (blur)="onClick($event)" />
4 <app (key.enter)="onEnter($event)" />
5 <app (key.down)="onEnter($event)" />
6 ...
```

LET'S GO

- Créer un composant qui contient un bouton
 - soit il affiche '-' soit '+'
- Créer un second composant qui affiche un bouton plus et moins
- créer un 3ème composant qui utilise le 2ème pour incrémenter ou décrémenter un compteur (initialisé à 0)

CONTROL FLOW

CONDITION

Pour afficher une condition dans le template on utilise @if, @else if, @else et l'écriture comme une condition js:

```
1 <div>
2   @if(maCondition === 0) {
3     <span>Je vaux 0</span>
4   } @else if (maCondition === 1) {
5     <span> Je vaux 1</span>
6   } @else {
7     <span> Je suis plus grand que 1</span>
8   }
9 </div>
```

CONDITION

On peut donner un alias à notre valeur:

```
1 <div>
2   @if(maCondition === 0; as isZero) {
3     <span>Je vaux 0: {{ isZero }}</span>
4   }
5 </div>
```

DÉFINIR UNE VARIABLE

Pour créer une variable dans le template: @let

```
1 <div>
2   @let isZero = true;
3   @if(isZero) {
4       <span>Je vaux 0</span>
5   }
6 </div>
```

BOUCLE

Pour créer une boucle dans le template: @for

- une fonction de tracking est obligatoire (objet, index,...)

```
1 <div>
2   @for (item of items; track $index) {
3       <span>{{item.id}}</span>
4   }
5
6 </div>
```

BOUCLE

@for met à disposition à l'intérieur du bloc plusieurs variables:

- \$index: index de l'élément courant
- \$count: nombre d'élément de la collection
- \$first: indique si c'est le premier élément
- \$last: indique si c'est le dernier élément
- \$even: indique si l'élément est pair dans la collection
- \$odd: indique si l'élément est impair dans la collection

Si plusieurs boucles sont imbriquées on peut

faire des alias de ces variables:

```
@for(item of items: track item.id; let id = $index;  
let even $event) {...}
```


BOUCLE

Si on veut afficher quelque chose si la liste est vide on peut utiliser @empty :

```
1  @for(item of items; track $index) {  
2      ...  
3  } @empty {  
4      <span>Aucun élément</span>  
5  }
```

SWITCH

Pour créer une contition de type switch dans le template: @switch, @case, @default:

```
1  <div>
2    @switch(maPropriete){
3      @case('57'){
4        <span>Moselle</span>
5      }
6      @case('974') {
7        <span>La Réunion</span>
8      }
9      @default {
10       <span>les autres</span>
11     }
12   }
13 </div>
```

LET'S GO

Dans un nouveau composant, créer une liste de personne. L'afficher dans une boucle for.

Si la personne est majeur, on affiche majeur à côté de son age, mineur sinon.

LES DIRECTIVES

DIRECTIVES

Les composants sont des directives avec une vue.

Elles vont permettre de "enrichir" un element html, et améliorer les compositions d'éléments.

Ils existent des directives structurelles qui vont directement modifier le DOM => par exemple, on a *ngIf, *ngFor, *ngSwitch. Mais sont remplacé par les controles flow (toujours présente dans le framework cependant)

DIRECTIVES

Elles utilisent comme des attributs html, sauf si elles ont des inputs et s'écriront ainsi comme un input

```
1 @Directive({
2     selector: 'app-couleur'
3 })
4 export class AppColorDirective {
5     couleur = input<'red' | 'blue' | 'green'>({alias: 'app-couleur'});
6     text = input<string>();
7
8     constructor(elementRef: ElementRef, render: Renderer2) {
9         effect(() => {
10             render.setStyle(
11                 this.elementRef.nativeElement,
12                 'backgroundColor',
13                 this.couleur()
14             )
15         });
16     }
17 }
18 ....
19
20 <span app-couleur="red" [text]=" 'toto' "></span>
```

LES PIPES

PIPE

Elles vont permettre la transformation de donnée pour l'**affichage**.

Il en existe des pré-inclus dans le framework:

- AsyncPipe : transforme une donnée asynchrone (promise, observable) en donnée lisible.
- CurrencyPipe: affiche les devises selon la langue
- DatePipe: affiche une date lisible depuis l'objet Date
- JsonPipe: affiche un objet au format json
-
- <https://angular.dev/guide/templates/pipes#built-in-pipes>

PIPE

Pour utiliser une pipe on va utiliser le caractère | dans le template:

```
1 <h1>{{ hier | date }}</h1>
```

On peut lui passer des paramètres:

```
1 <h1>{{ hier | date: 'hh:mm' }}</h1>
```

Et les chainer

```
1 <h1>{{ hier | date | uppercase }}</h1>
```

PIPE

Pour créer une pipe on utilise le décorateur @Pipe sur une classe qui implémente PipeTransform

```
1  @Pipe({
2    name: 'uppercase'
3  })
4  class toto implements PipeTransform{
5    transform(value: string, ...args: any[]) {
6      return value.toUpperCase();
7    }
8  }
```

LET'S GO

- Implémenter une pipe qui met en majuscule la première lettre de la valeur d'entrée
- L'ajouter sur le nom et prénom du composant précédent

LES MODULES

MODULE

Permet de regrouper, déclarer des composants non standalone et de les exporter (= rendre visible à l'extérieur du module).

```
1  @NgModule({
2    imports: [AppModule],
3    declarations: [TotoComponent],
4    exports: [TotoComponent, AppModule]
5  })
6  export class TestModule {}
7
8  @NgModule({
9    imports: [TestModule],
10   declarations: [TataComponent],
11   exports: [TataComponent]
12 })
13 export class Test2Module {}
```

MODULE

Un composant non standalone doit avoir standalone: true dans les metadata (depuis v16, standalone: false par défaut, true par défaut depuis v19).

```
1  @NgModule({
2    imports: [AppModule],
3    declarations: [TotoComponent],
4    exports: [TotoComponent, AppModule]
5  })
6  export class TestModule {}
7
8  @NgModule({
9    imports: [TestModule],
10   declarations: [TataComponent],
11   exports: [TataComponent]
12 })
13 export class Test2Module {}
```

LES SERVICES

SERVICE

C'est une classe "utilitaire", avec un décorateur @Injectable

```
1  @Injectable()  
2  export class MonService {  
3  
4  }
```


SERVICE

Par défaut, un service est non "provider",
on va pouvoir définir la porter de plusieurs façon, la première est
de le provider à la racine de l'application

```
1  @Injectable({  
2      providedIn: 'root',  
3  })  
4  export class MonService {  
5  
6  }
```

SERVICE

Dans un composant et ses fils,

dans un module (même mot clé que pour le composant)

```
1  @Composant( {  
2      ...,  
3      providers: [MonService]  
4  } )  
5  export class MonComposant {  
6  
7  }
```

SERVICE

Manuellement à la racine du projet

```
1 export const appConfig: ApplicationConfig
2   = {
3     providers: [MonService]
4   };
```

SERVICE

- Pour l'utilisation, plusieurs façon de faire
 - depuis le constructeur
 - avec la fonction inject

```
1 export class AppComponent {  
2     private readonly monService = inject(MonService);  
3  
4     constructeur(private readonly monService: MonService) {  
5     }  
6 }
```

HTTP



HTTP

Angular fournit une abstraction au API du navigateur pour faire une requête:

Pour l'initialiser il faut injecter le service `http: provideHttpClient()` (avant on importait le module `HttpClientModule`)

Pour utiliser ce client `http`, il faut injecter le service `HttpClient` dans notre composant:

```
1 private http = inject(HttpClient);
```

HTTP

On va ensuite pouvoir faire des appels réseau !

```
1 getMaData(): Observable<Data> {  
2     this.http.get('/url/serveur/id');  
3     this.http.post('/url/serveur', { body: ''});  
4     this.http.put('/url/serveur', { body: ''});  
5     this.http.delete('/url/serveur/id');  
6 }
```

HTTP

C'est pas tout, http est asynchrone, il faut écouter le retour !

```
1  getMaData(): void {  
2      this.http  
3          .get( '/url/serveur/id' )  
4          .subscribe(data => {  
5              console.log(data);  
6          })
```

Ici on utilise RXJS ! On pourrait avoir un module complet tellement c'est complexe.

HTTP

Plutôt que de faire une souscription de l'événement à la main, on peut directement utiliser la pipe async dans le template

```
1  getMaData(): void {  
2      this.http.get( '/url/serveur/id' );  
3  }  
4  ///  
5  @if( getMaData() | async; as data ) {  
6      {{ data }}  
7  }
```

HTTP

on peut aussi convertir l'observable en signal avec toSignal

```
1  getMaData(): Signal<Data> {  
2      return toSignal(this.http.get( '/url/serveur/id' ));  
3  }
```

HTTP - INTERCEPTEUR

Permet d'agir sur les données envoyées vers (et du) serveur

Pour ajouter un nouveau intercepteur, modifier la config du client

http (ici l'intercepteur c'est auth)

```
1 provideHttpClient(  
2     withInterceptors([auth])  
3 )
```

HTTP - INTERCEPTEUR

L'intercepteur est une fonction (il existe aussi une version class + décorateur, mais déprécié aujourd'hui)

```
1  const auth =  
2      (req: HttpRequest<unknown>, next: HttpHandlerFn) => {  
3          req.  
4          return next(req);  
5      };
```

LET'S GO

- Créer un service qui appelle l'api <https://api.chucknorris.io/jokes/random>
- afficher l'icone et la value dans un composant
- ajouter un intercepteur qui log l'appel (l'url et la méthode)

LES ROUTES

ROUTING

- Elles sont défini dans un objet qu'on injecte dans un service
 - `provideRouter(routes)`
- `routes` contient un tableau de "chemin" :

```
1  export const routes: Routes = [{  
2    path: 'route1',  
3    component: TotoComponent  
4  }];
```

ROUTING

- On va via la route découper notre application afin qu'elle charge au besoin

```
1 export const routes: Routes = [{  
2   path: 'route1',  
3   loadComponent: import( '/path/toto.component' )  
4     .then(component => component.TotoComponent)  
5 }];
```


ROUTING

- Sur une route on va pouvoir ajouter des "guard" qui vont restreindre l'accès:

```
1  const isConnecte = ()  
2      => inject(AuthService).isConnecte  
3  
4  export const routes: Routes = [{  
5      path: 'route1',  
6      canActivate: [isConnecte]  
7      loadComponent: import( '/path/toto.component' )  
8          .then(component => component.TotoComponent)  
9  }];
```

ROUTING

- Sur une route on va pouvoir précharger des données avec resolve :

```
1  const isConnecte = ()
2      => inject(AuthService).isConnecte
3
4  export const routes: Routes = [{
5      path: 'route1',
6      resolve: {
7          contrat: monResolver
8      }
9      loadComponent: import( '/path/toto.component' )
10         .then(component => component.TotoComponent
11     } ];
```

ROUTING

- Un resolver s'écrit de la façon suivante:

```
1  const monResolver =  
2  (  
3    route: ActivatedRouteSnapshot,  
4    state: RouterStateSnapshot  
5  )  
6  => {  
7    return inject(Data).getMaData();  
8  }
```

ROUTING

- Pour savoir on injecter nos composants de route, sur le parent on va devoir ajouter la balise "router-outlet"

```
1  <h1>Mon App<h1>  
2  
3  </h1><app-bar></app-bar>  
4  
5  <router-outlet />
```

NAVIGATION

Pour naviguer:

- soit on utilise le service Router
 - avec la méthode `navigateByUrl` qui prend le chemin depuis la base
 - avec la méthode `navigate` qui prend un tableau de chemin
- soit via lien html "a" + directive `[routerLink]`

```
1 <a [routerLink]="/path1">Allez sur page 1</a>
```

LET'S GO

- Créer une application avec deux routes
 - une route qui affichera une liste des joueurs
 - une route qui affichera la liste des matchs
- Ajouter un menu pour pouvoir naviguer entre les routes
- Elle appelle votre backend express pour avoir la liste des joueurs (pour l'instant retourner une liste vide pour les matchs (@empty 🤗))

CAN MATCH

```
1 export const routes: Routes = [  
2   {  
3     path: 'animal',  
4     canMatch: [() => inject(AnimalService).isCat],  
5     component: CatComponent  
6   },  
7   {  
8     path: 'animal',  
9     canMatch: [() => inject(AnimalService).isCat],  
10    component: CatComponent  
11  },  
12  {  
13    path: '',  
14    redirectTo: '/animal'  
15  },  
16  {  
17    path: '**',  
18    component: NotFound404Component  
19  }  
20 ];  
21 ];
```

PARAMETRES

Pour créer un paramètre depuis la route: on utilise ":" + nom du paramètre

```
1 export const routes: Routes = [  
2   {  
3     path: ':id',  
4     component: Component  
5   }  
6 ];  
7
```


PARAMETRES

Pour récupérer la valeur : on va utiliser les inputs simplement !
Mais il faut activer une config, withComponentInputBinding :

```
1 provideRouter(appRoutes, withComponentInputBinding())
```

Sinon, on injecte ActivatedRoute et lit la propriété "data" sur ce service.
(dispo seulement après ngOnInit)

Avec ça on pourra récupérer:

- query parameters (?XXXXX=toto&YYY=tata)
- path parameters (/contrat/123)
- static route data (data sur la route)
- data from resolvers

LES FORMULAIRES

FORMULAIRES

Il existe deux manières d'écrire les formulaires

- Template driven
 - Utilise le module FormsModule
 - Utilise des directives
 - Toute la logique est coté html
- Reactive Forms
 - Utilise le module ReactiveFormsModule
 - Logique dans le .ts
 - Plus verbeux

TEMPLATE DRIVEN FORMS

On va déclarer référence sur un composant (avec #):

```
1 <form (ngSubmit)="formName.value">
2     <input #formName />
3     <button type="submit">Send</button>
4 </form>
```

TEMPLATE DRIVEN FORMS

Utilisation de ngModel, pour mieux controler les champs

```
1 <form (ngSubmit)="formName.value">
2   <input
3     name="nom"
4     ngModel
5     #nom="ngModel"
6     required />
7   <button
8     type="submit"
9     [disabled]="nom.errors?.required" >
10     Send
11   </button>
12 </form>
```

TEMPLATE DRIVEN FORMS

On va aussi pouvoir ajouter une variable directement sur form et récupérer l'état/valeur global.

`monFormulaire.value` retourne un objet JSON avec les clés/valeur: nom du champs / champs.value

```
1 <form
2     #monFormulaire="ngForm"
3     (ngSubmit)="sendToBackend(monFormulaire.value)">
4     <input
5         name="nom"
6         ngModel
7         #nom="ngModel"
8         required />
9 </form>
```

TEMPLATE DRIVEN FORMS

Sur le controleur (nom dans l'exemple précédent), on va avoir des attribut d'état utile pour la validation:

- pristine: l'utilisateur a modifié le champs
- dirty: l'inverse de pristine
- valid: respect les validations (required,length, pattern,...)
- invalid: l'inverse de valid
- touched: l'utilisateur à interagie avec le champs (click, focus,...)
- untouched: l'inverse de touched

TEMPLATE DRIVEN FORMS

Pour modifier le style de nos composants en fonction de l'état, angular va ajouter des classes:

- .ng-valid /.ng-invalid
- .ng-pristine /.ng-dirty
- .ng-touched /.ng-untouched

LET'S GO

Créer un formulaire avec

- nom
- prenom
- postes: champs select
- un bouton de validation
 - désactivé seulement si tout les champs sont valide
 - à la validation: affiche les données des champs sous le formulaire.
- Si les champs sont "touched" et non valide, on affiche une erreur

REACTIVE FORMS

- On va pouvoir déplacer la logique dans le composant (et laisser le html pour la vue)
- On va créer un objet `FormGroup` qui contiendra la déclaration des noms, des validations, valeur initiales, structure,...
- un champs est déclarer avec un `FormControl`
- `FormGroup` et `FormControl` possède les mêmes api que sur un `formmodule` (`valid`, `dirty`, ...)

```
1 monFormulaire = new FormGroup({  
2     nom: new FormControl('')  
3 });
```

REACTIVE FORMS

une fois le formgroup créé, il faut le déclarer sur le formulaire dans le template, on va utiliser la directive (du module ReactiveFormsModule): formGroup

Sur le champs on va utiliser la directive formControlName

```
1  <form
2      [formGroup]="monFormulaire"
3      (ngSubmit)="monFormulaire.value">
4
5      <input formControlName="nom" />
6      <button type="submit">Send</button>
7  </form>
```

REACTIVE FORMS

Pour ajouter un controle sur nos champs, on va utiliser les `Validators`.

Angular fourni les plus classique (required, min, max, length,...)

```
1 monFormulaire = new FormGroup({
2     nom: new FormControl(
3         '', [
4             Validators.required,
5             Validators.minLength(2)
6         ]
7     )
8 } );
```

REACTIVE FORMS

Pour récupérer la valeur du formulaire :

```
form.value
```

Attention, si un champs est désactivé, la valeur ne sera pas incluse,
pour ça il faut utiliser `form.getRawValue()`

Pour récupérer la valeur d'un champs en particulier:

```
form.controls.XXX.value
```

REACTIVE FORMS

Avec le formGroup/formControl, on va pouvoir "écouter" les changements de valeur du formulaire depuis le composant :

Attention il faut penser (et c'est valable pour tout observable) à supprimer l'écoute à la destruction du composant, au risque d'avoir des fuites mémoires, voir plus grave !

```
1  this.form.controls.nom.valueChanges
2  .pipe(takeUntilDestroyed()) //Stop l'écoute quand le
   composant est détruit
3  .subscribe(value => {
4      console.log(`Nouvel valeur: ${value}`);
5  });
```

LET'S GO

Convertir le formulaire précédent en reactive form

ANGULAR MATERIAL

LIBRAIRIE DE COMPOSANT

- <https://material.angular.io/>
- Implémente la spec UI Material Design
 - Règle de design chez Google
 - Actuellement en à la v3 de cette spec

ANGULAR MATERIAL

Pour ajouter angular material à notre application :

- `ng add @angular/material`

```
✓ Choose a prebuilt theme name, or "custom" for a custom theme: Azure/Blue
https://material.angular.io?theme=azure-blue]
✓ Set up global Angular Material typography styles? Yes
✓ Include the Angular animations module? Include and enable animations
UPDATE package.json (1104 bytes)
✓ Packages installed successfully.
UPDATE src/app/app.config.ts (421 bytes)
UPDATE angular.json (2972 bytes)
UPDATE src/index.html (516 bytes)
UPDATE src/styles.scss (181 bytes)
```

ANGULAR MATERIAL

Pour ajouter angular material à notre application :

- `ng add @angular/material`

```
✓ Choose a prebuilt theme name, or "custom" for a custom theme: Azure/Blue
https://material.angular.io?theme=azure-blue]
✓ Set up global Angular Material typography styles? Yes
✓ Include the Angular animations module? Include and enable animations
UPDATE package.json (1104 bytes)
✓ Packages installed successfully.
UPDATE src/app/app.config.ts (421 bytes)
UPDATE angular.json (2972 bytes)
UPDATE src/index.html (516 bytes)
UPDATE src/styles.scss (181 bytes)
```

ANGULAR MATERIAL

Exemple: Composant card

API reference for Angular Material card

```
import {MatCardModule} from '@angular/material/card';
```

Card with actions alignment option

[↔](#) [<>](#) [✎](#)

HTML

TS

```
<mat-card appearance="outlined">
  <mat-card-header>
    <mat-card-title> Australian Shepherd</mat-card-title>
    <mat-card-subtitle>Herding group</mat-card-subtitle>
  </mat-card-header>
  <mat-card-actions>
    <button mat-button>Learn More</button>
  </mat-card-actions>
</mat-card>
```



LET'S GO


Utiliser le composant card sur votre composant avec la liste de personne

INJECTION TOKEN

INJECTION TOKEN

Permet d'utiliser de partager de façon statique une valeur dans toute l'application.

Pour créer un token :



```
1 export const MA_VALEUR = new InjectionToken<string>('MA_VALEUR');
```

INJECTION TOKEN

Maintenant qu'on a notre token, il faut lui "provider" une "value". Dans les propriétés "providers" (comme les services), on passe un objet provide/useValue de la façon suivante :

```
1 providers: [  
2     {  
3         provide: MA_VALEUR,  
4         useValue: '123'  
5     }  
6 ]
```


INJECTION TOKEN

Cette syntaxe permet aussi de manipuler des services ! Très utile pour instancier un service abstrait par exemple.

A la place de "useValue", on peut aussi avoir :

- useClass: prend en paramètre une classe injectable, qu'angular ce chargera d'instancier.
- useExisting: remplace l'instance du "provide" par celle passé à useExisting (déjà providé en root par exemple)
- useFactory: instancie l'objet provide avec une méthode

LES TESTS

TESTS

Pour lancer les tests on va utiliser la commande suivante:




```
1 ng test
```

TESTS

- Par défaut, angular utilise jasmine comme framework de test
- en complément, pour lancer les tests dans un "vrai" navigateur, on utilise Karma
 - Karma est un outil en fin de vie, les développeurs de google sont en train de faire un étude pour trouver un outil remplaçant. En attendant ça marche bien ;)

TESTS


Rappel : un test js s'écrit à l'aide de fonctions imbriquées :



```
1 describe('mon scénario', () => {
2
3   beforeEach(() => {
4     //run commun config
5   })
6
7   it('mon test 1', () => {
8     expect(...).ToBe(...);
9   })
10 })
```

TESTS


Pour lancer un test on va avoir besoin de "monter" un composant de test. Angular fourni (un framework complet je vous disais!) une api de test: `TestBed`



```
1 beforeEach(async () => {  
2     await TestBed.configureTestingModule({  
3         import: [MonComposantATester],  
4         providers: [MonServiceDeTest]  
5     }).compileComponents();  
6 })
```

TESTS


Une fois le composant configuré, on le crée et on récupère une instance de `ComponentFixture`. Cet objet permet de contrôler notre composant.



```
1 let fixture: ComponentFixture<MonComposant>;
2 let component: MonComposant;
3
4 beforeEach(async () => {
5     await TestBed.configureTestingModule(...).compileComponents(
6
7     fixture = TestBed.createComponent(MonComposant);
8     component = fixture.componentInstance;
9     fixture.detectChanges();
10 })
```

TESTS


On va pouvoir interroger les composants sur le dom, on va pouvoir ainsi écrire nos tests et vérifier que tout est conforme.



```
1 it('should have a button', () => {
2   const button = fixture.debugElement.query(By.css('button'))
3
4   expect(button).not.toBe(undefined);
5 });
```


TESTS

Pour tester une méthode asynchrone, on utilise done



```
1 it('should have a button', (done) => {
2   component.maMethodeAsynchrone().subscribe(() => {
3     expect(...).toXXX(...);
4     done();
5   })
6 });
```

TESTS


Mocker un service afin d'isoler notre composant:

On va utiliser les mocks de jasmine + syntaxe provide/useValue d'angular pour donner une version modifier du service de base.

```
1  const monServiceSpy: jasmine.SpyObj<MonService>;
2
3  beforeEach(async () => {
4    monServiceSpy
5      = jasmine.createSpyObj<MonService>([ 'methodeAMocker' ] );
6    await TestBed.configureTestingModule({
7      imports: [AppComponent],
8      providers: [
9        { provide: MonService, useValue: monServiceSpy}
10     ]
11   }).compileComponents();
```

TESTS

Et on utilise le mock au besoin :



```
1 it('should work', (done) => {
2     //Sans cette ligne, par défaut on appelle une méthode vide
3     monServiceSpy.methodeAMocker.and.returnValue(false);
4     component.maMethodeQuiAppelleMonService();
5
6     expect(monServiceSpy.methodeAMocker.calls.count()).toBe(1)
7 });
```

LET'S GO

- Ajouter un test qui vérifie que le bouton est actif quand le formulaire est rempli
 - `control.setValue("XXX")` pour remplir le formulaire programatiquement.

QUESTIONS ?

TP FINAL

<https://docs.google.com/document/d/132e9fdUvAuhU3XUWtYV9DYMvribMhwftletdLQgVp8s/edit?usp=sharing>