FRAMEWORK ANGULAR

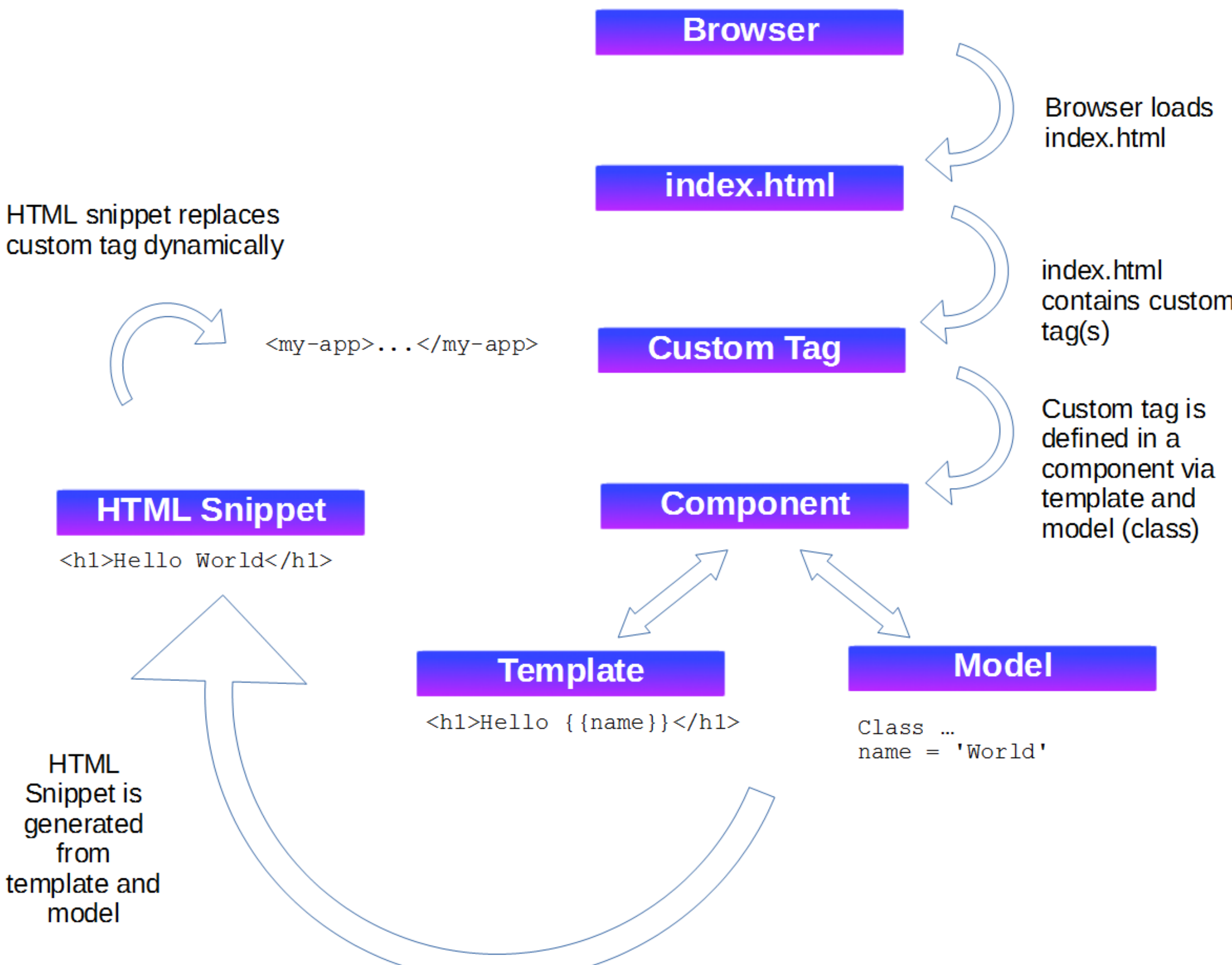
Utilisation de Framework

Notion de composant web

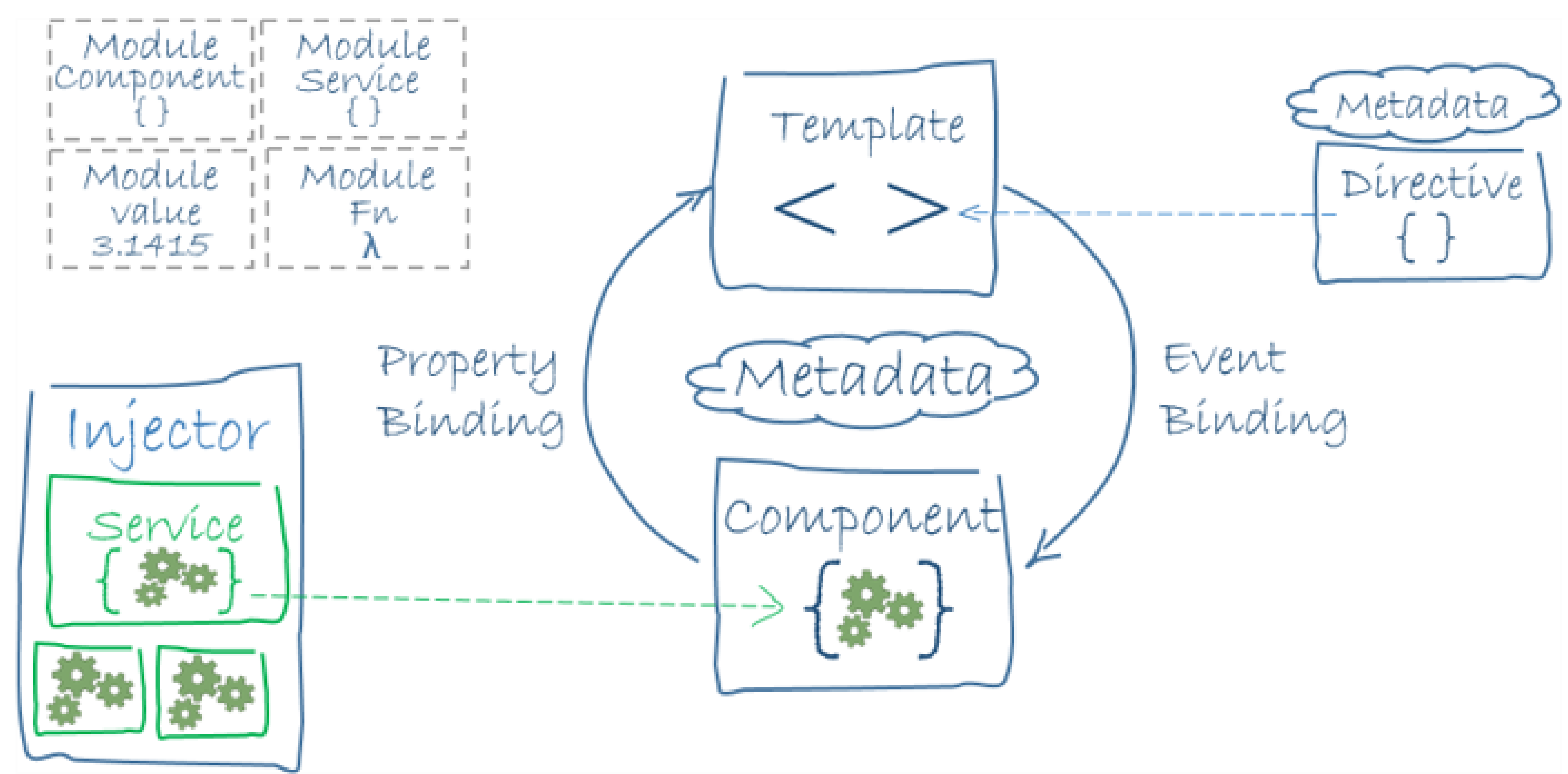
Concept d’Angular

ANGULAR

→ Angular s'appuie fortement sur la notion de web component



ANGULAR



ANGULAR : CONCEPTS

→ **Components** : élément de base d'une application Angular, permet de définir la manière dont l'utilisateur interagit avec la vue.

→ **Templates** : rendu HTML du composant sur une page.

→ **Data bindings** : relation entre les données d'un composant (modèle) et les valeurs affichées dans le template.

→ **Metadata :** information permettant de relier des éléments Angular (template et component par exemple pour former la vue)

→ **Component interaction :** lien entre les différents composants (échange d'information)

ANGULAR : CONCEPTS

→ **Dependency injection / service** : implémentation du pattern IoC (inversion des contrôles) afin de gérer les dépendances d'une application.

→ **Routing** : gérer l'aspect navigation d'une SPA.

→ **Forms** : gestion de la saisie utilisateur.

→ **Pipe :** transformation de la valeur d'un élément avant de l'afficher dans la vue (e.g. date).

→ **Modules :** organisation d'une application en bloc fonctionnel.

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d’Angular

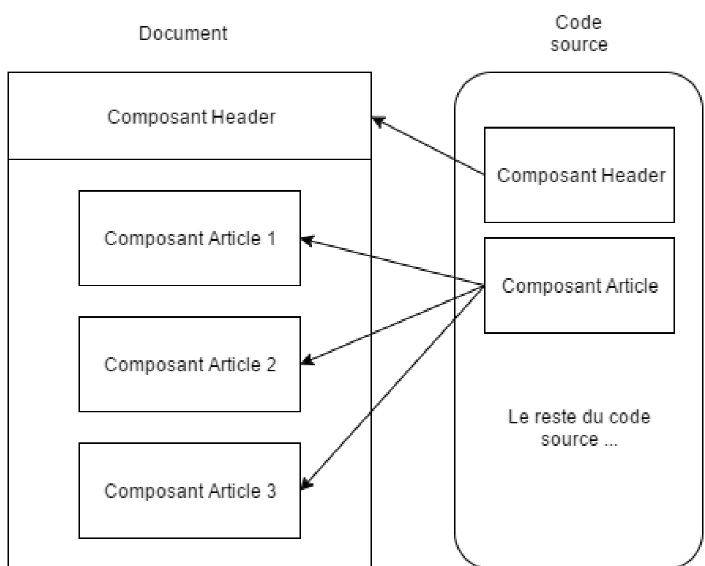
Component / Template

ANGULAR : COMPOSANT

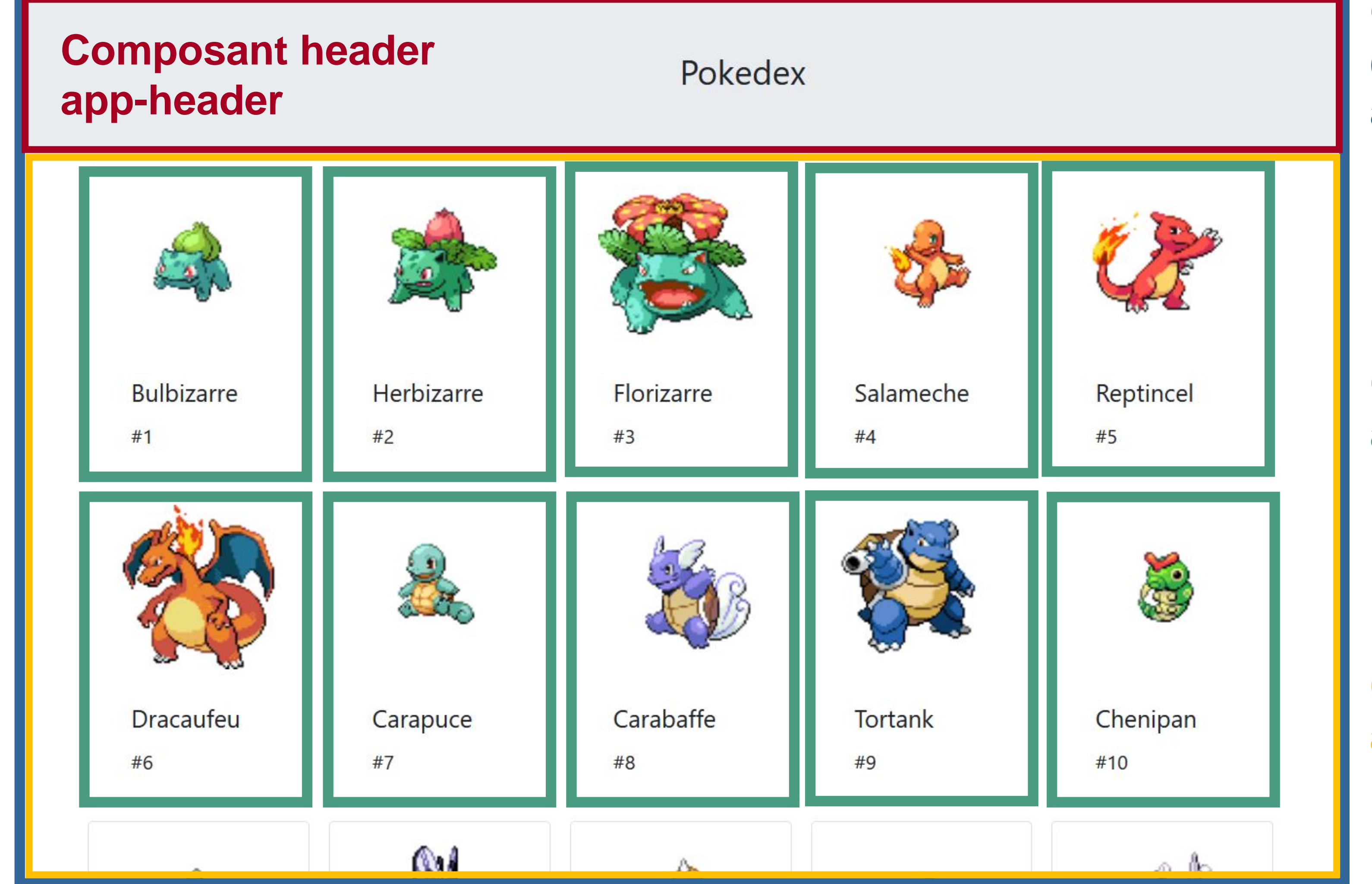
→ Structure fondamentale d’Angular (et d’autres frameworks et Web Component).

→ Une application est découpée en composant qui peuvent contenir eux-mêmes d’autres composants.

→ Avantages : réutilisation des composants et découpage logique.



ANGULAR : COMPOSANT



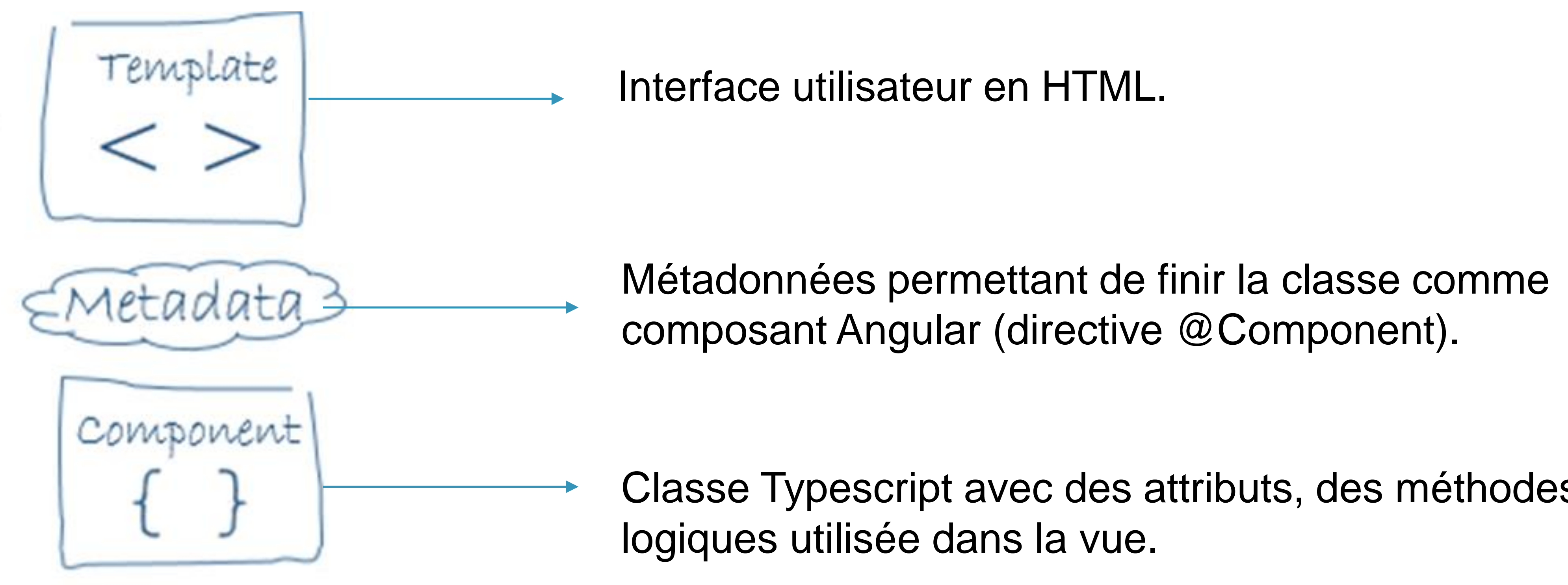
Composant principal
(app component)
app-root

Composant pokemon
app-pokemon

Composant pokedex
app-pokedex

ANGULAR : COMPOSANT

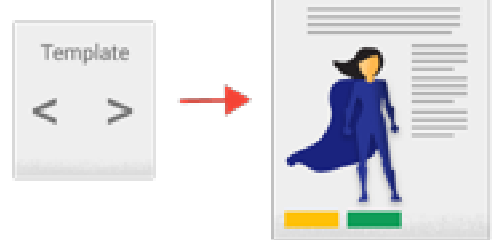
→ **Un composant Angular ?**



ANGULAR : TEMPLATE

→ Correspond à la vue du composant dans la page de l’application Angular.

→ Possibilité d’utilisé des directives, de déclencher un appel d’évènement, d’afficher les données mise à jour des composants (databinding), d’instancier d’autres composants et bien d’autres.



ANGULAR : PREMIER COMPOSANT

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

ANGULAR : PREMIER COMPOSANT

→ Fichier app.component.ts

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

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

export class AppComponent {
  title = 'Premier exemple de cours';
}
```


→ Fichier app.component.html

```
<header class="container-fluid bg-light p-2">
  <h1>{{title}}</h1>
</header>
```

ANGULAR : PREMIER COMPOSANT

Permet d'importer le décorateur Component

→ Fichier app.component.ts

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

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

export class AppComponent {
  title = 'Premier exemple de cours';
}
```

Décorateur **@Component** :
permet de déclarer le composant

Classe liée au composant

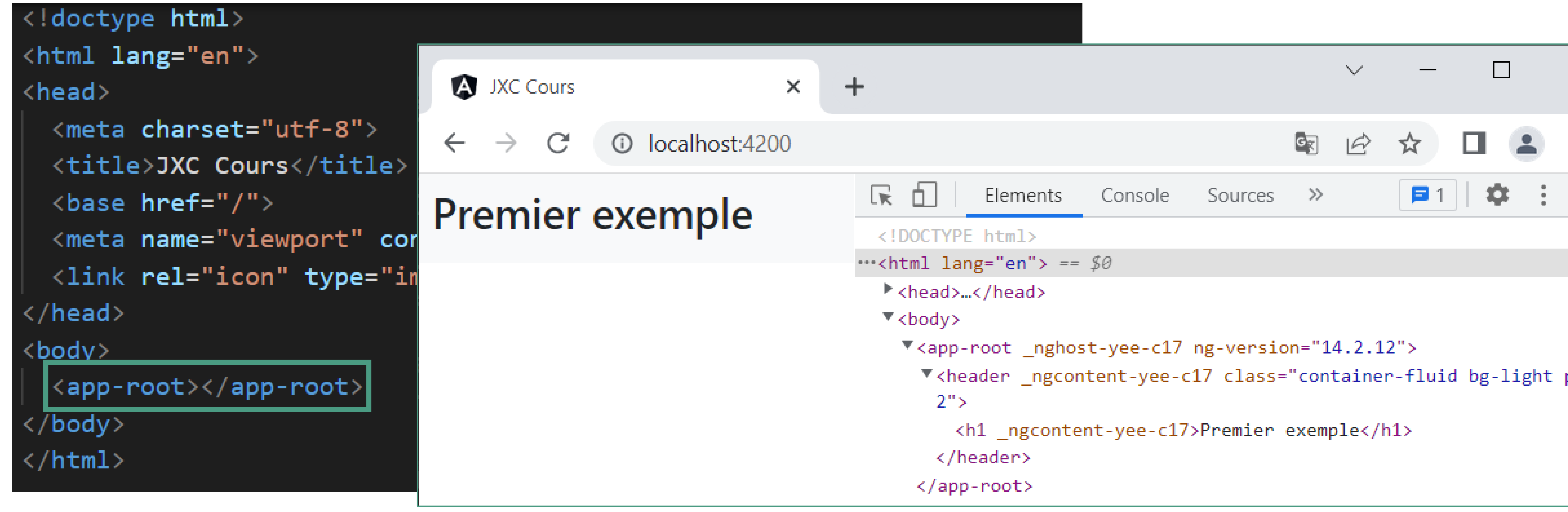
→ Fichier app.component.html

```
<header class="container-fluid bg-light p-2">
  <h1>{{title}}</h1>
</header>
```

Interpolation du texte

ANGULAR : PREMIER COMPOSANT

→ Fichier **index.html** avec utilisation de notre composant racine (selecteur **app-root**).



ANGULAR : PREMIER COMPOSANT

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages.

ANGULAR : COMPOSANT

→ Possibilité de créer **des classes et des interfaces** pour les données afin de les utiliser dans les composants.

```
export interface IPersonnage {
    id: number;
    name: string;
}
```

CodeCours > jxc-cours > src > app > personnages > TS personnages.component.ts >

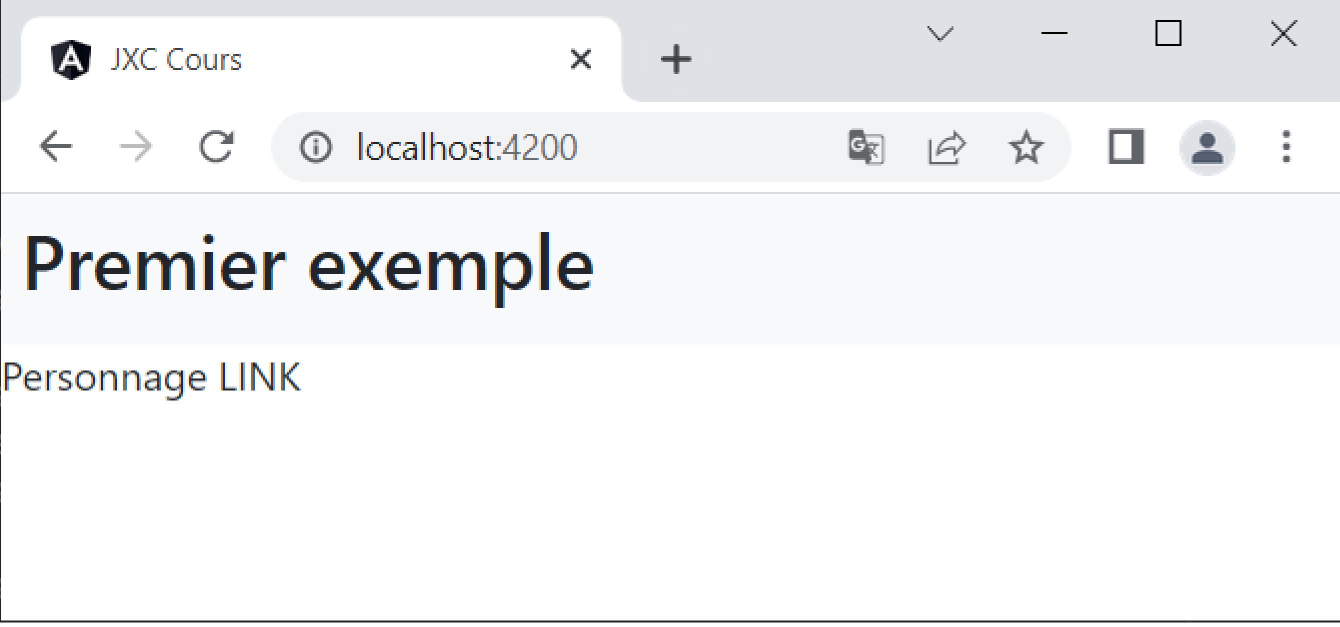
```
1  import { Component, OnInit } from '@angular/core';
2  import { IPersonnage } from './IPersonnage';
3
4  @Component({
5    selector: 'app-personnages',
6    templateUrl: './personnages.component.html',
7    styleUrls: ['./personnages.component.css']
8  })
9  export class PersonnagesComponent implements OnInit {
10   hero: IPersonnage = {
11     id: 1,
12     name: 'Link'
13   }
14   constructor() { }
15   ngOnInit(): void {
16   }
17 }
```

ANGULAR : COMPOSANT

→ Possibilité de créer **des classes et des interfaces** pour les données afin de les utiliser dans les composants.

```
export interface IPersonnage {
    id: number;
    name: string;
}
```

```
projetJXC > src > app > personnages > <> personnages.component.html > ..
1    <p>Personnage {{perso.name | uppercase}}</p>
```

ANGULAR : PREMIER COMPOSANT

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages.

→ Simuler des données d’un back pour les afficher dans le template de Personnages.

ANGULAR : COMPOSANT

→ Utilisation d'une liste de personnages dans le composant.

→ Création d'une constante (liste de personnages) pour simuler la récupération des données depuis un serveur.

CodeCours > jxc-cours > src > app > TS mock-personnages.ts > [@] PERSONNAGES

```
import { IPersonnage } from "./personnages/IPersonnage";

export const PERSONNAGES: IPersonnage[] = [
    {id:1, name: 'Link'},
    {id:2, name: 'Zelda'},
    {id:3, name: 'Revali'},
    {id:4, name: 'Urbosa'},
    {id:5, name: 'Sidon'},
    {id:6, name: 'Mipha'}
]
```

ANGULAR : COMPOSANT

→ Utilisation d'une liste de personnages dans le composant.

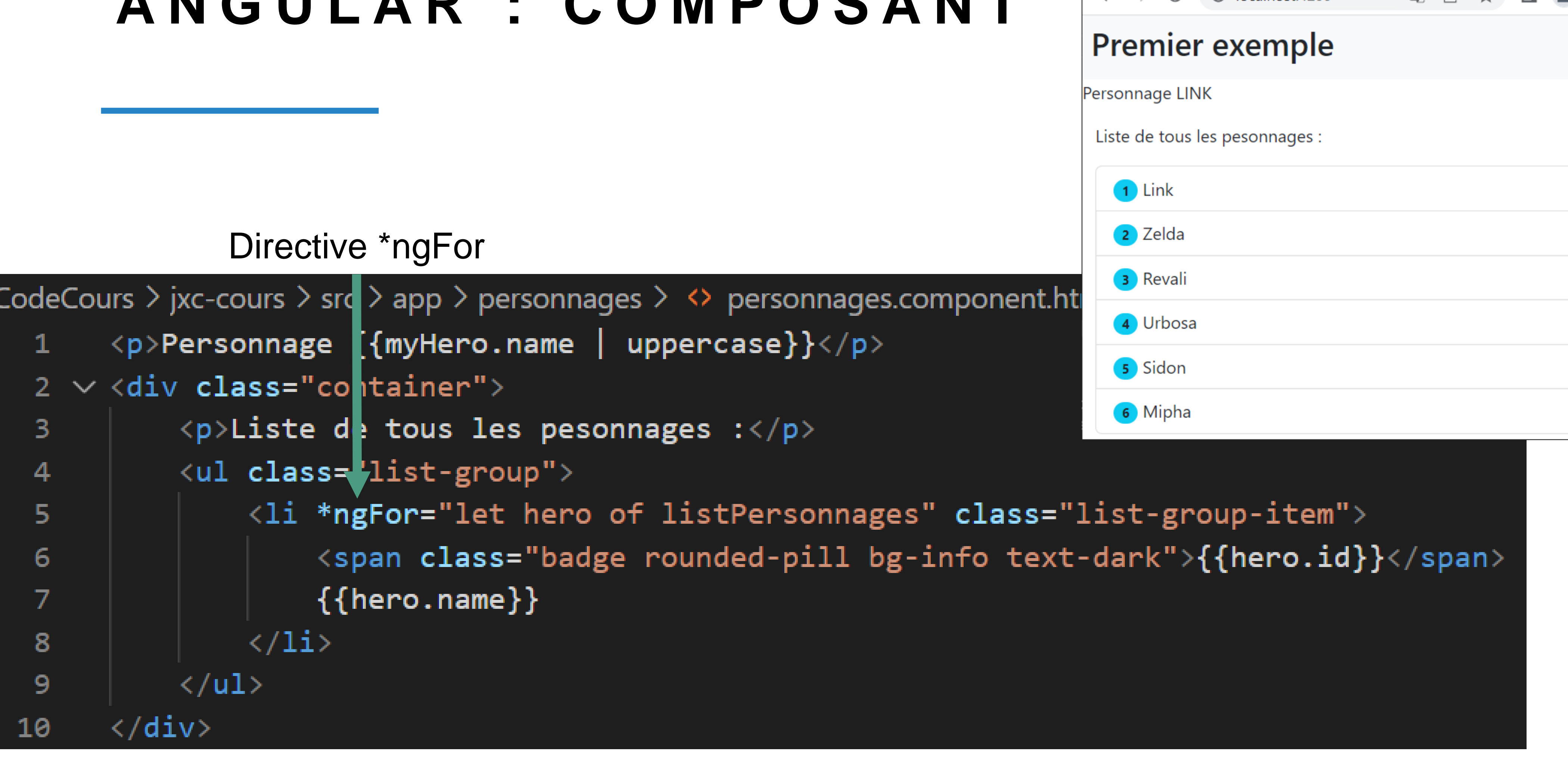
→ Création d'une constante (liste de personnages) pour simuler la récupération des données depuis un serveur.

CodeCours > jxc-cours > src > app > personnages > TS personnages.component.ts >

```
1   import { Component, OnInit } from '@angular/core';
2   import { PERSONNAGES } from '../mock-personnages';
```

```
9   export class PersonnagesComponent implements OnInit {
10
11    listPersonnages = PERSONNAGES;
12    myHero = this.listPersonnages[0];
```

ANGULAR : COMPOSANT



ANGULAR : CYCLE DE VIE

Cycle de vie des composants

Possibilité d'intercepter les différentes étapes du cycle de vie d'un composant (hook).

→ **ngOnChanges() :** appelée à la création du composant (après le constructeur), puis à chaque changement d'un attribut scalaire décoré par **@Input**.

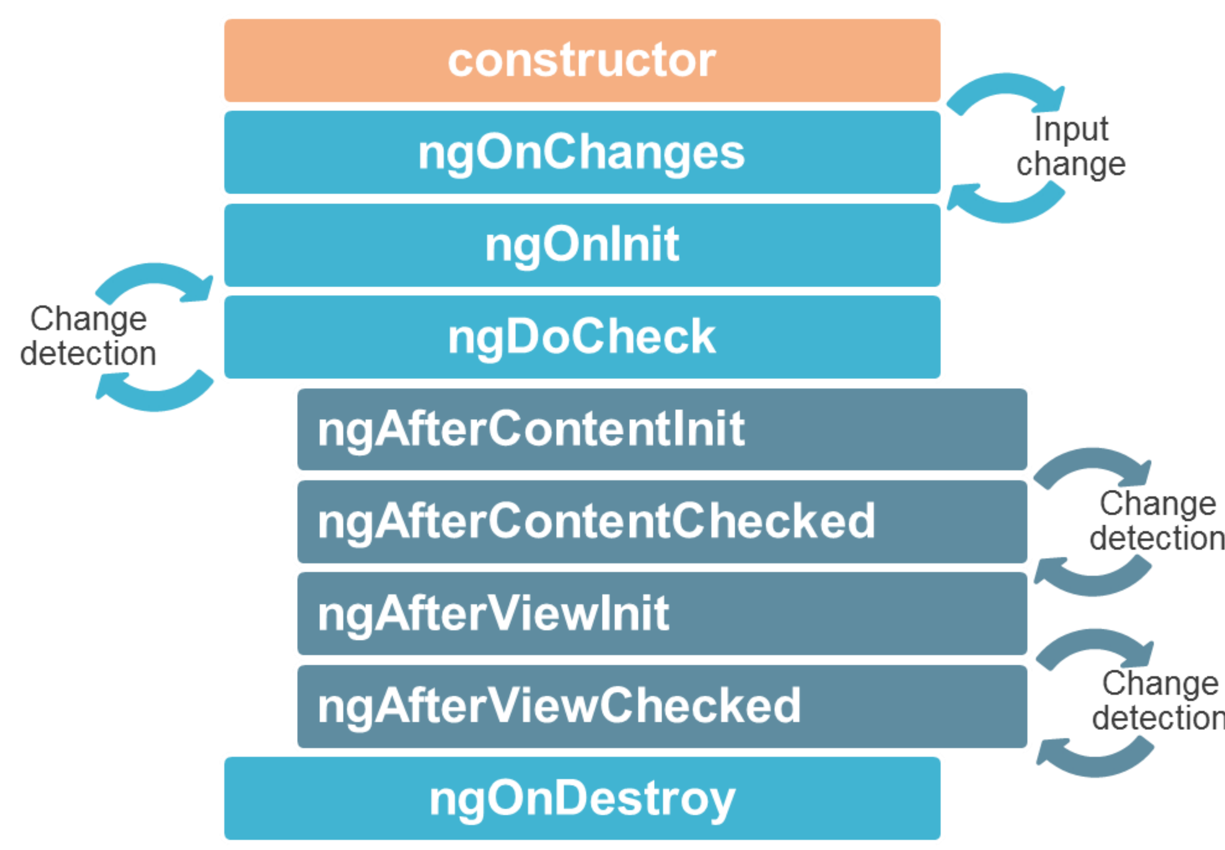
→ **ngOnInit() :** appelée une seule fois lors de la création d'un composant juste après le premier appel de *ngOnChange* (souvent appel aux API pour récupérer les données).

→ **ngDoCheck() :** mise en œuvre pour connaître les changements des valeurs internes d'objets ou de listes (ceux non identifiables par *ngOnChanges*)

→ **ngOnDestroy() :** appelée juste avant que le composant soit désalloué.

→ Et d'autres : *ngAfterContentInit*(), *ngAfterContentChecked*(), etc.

ANGULAR : CYCLE DE VIE



ANGULAR : VIEW ENCAPSULATION

Gestion de l'encapsulation

Possibilité de spécifier une *View Encapsulation* dans un composant.

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



```
@Component({
  selector: 'app-personnages',
  templateUrl: './personnages.component.html',
  styleUrls: ['./personnages.component.css'],
  encapsulation: ViewEncapsulation.None
})
```

→ **None** : aucune encapsulation de style réalisée, le style d'un composant est global à toute l'application.

→ **Emulated** : le Shadow DOM n'est pas utilisé mais une encapsulation émulée est fait pour les styles de composants (permet de limiter les styles des composants)

→ **ShadowDom** : utilisation du shadow DOM du navigateur (rendu plus rapide mais attention au support des navigateurs).

ANGULAR : VIEW ENCAPSULATION

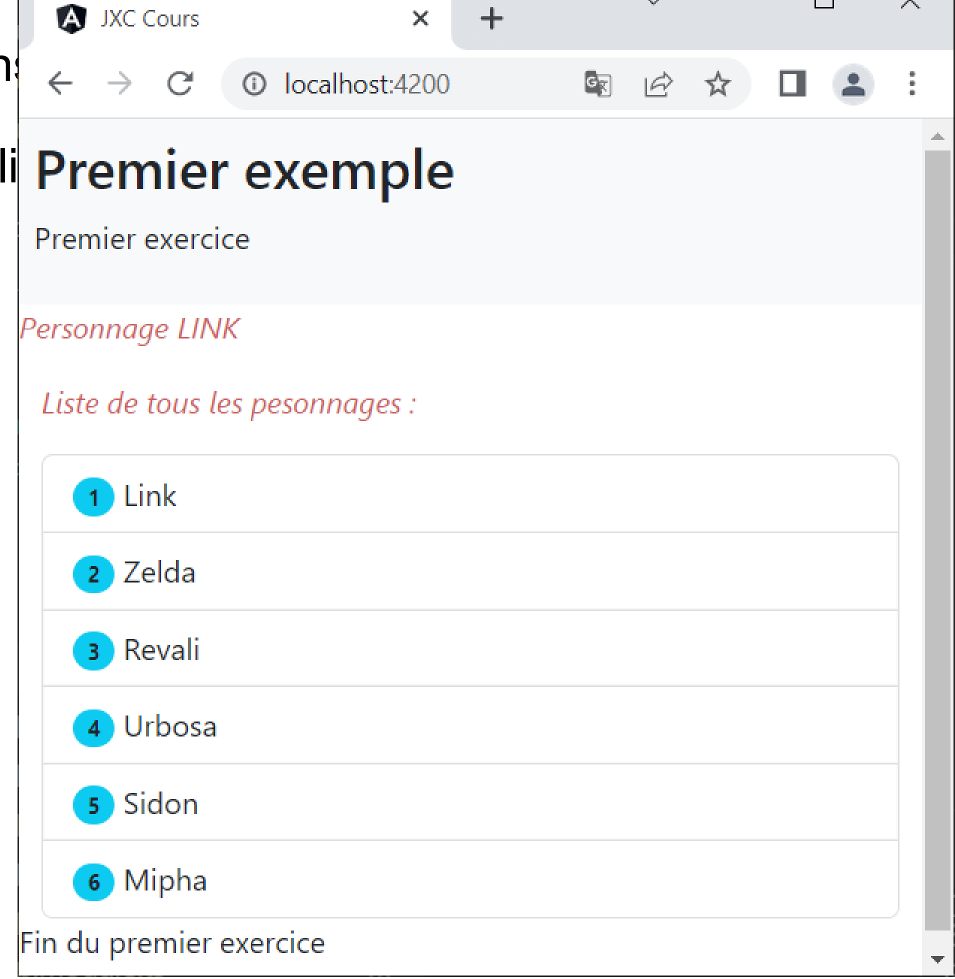
→ **Exemple :** ajout d’une balise <p> dans

ajout d’un style pour la bali

→ Pas de modification de **Personnages**.

→ Le style de <p> est celui défini dans Personnages

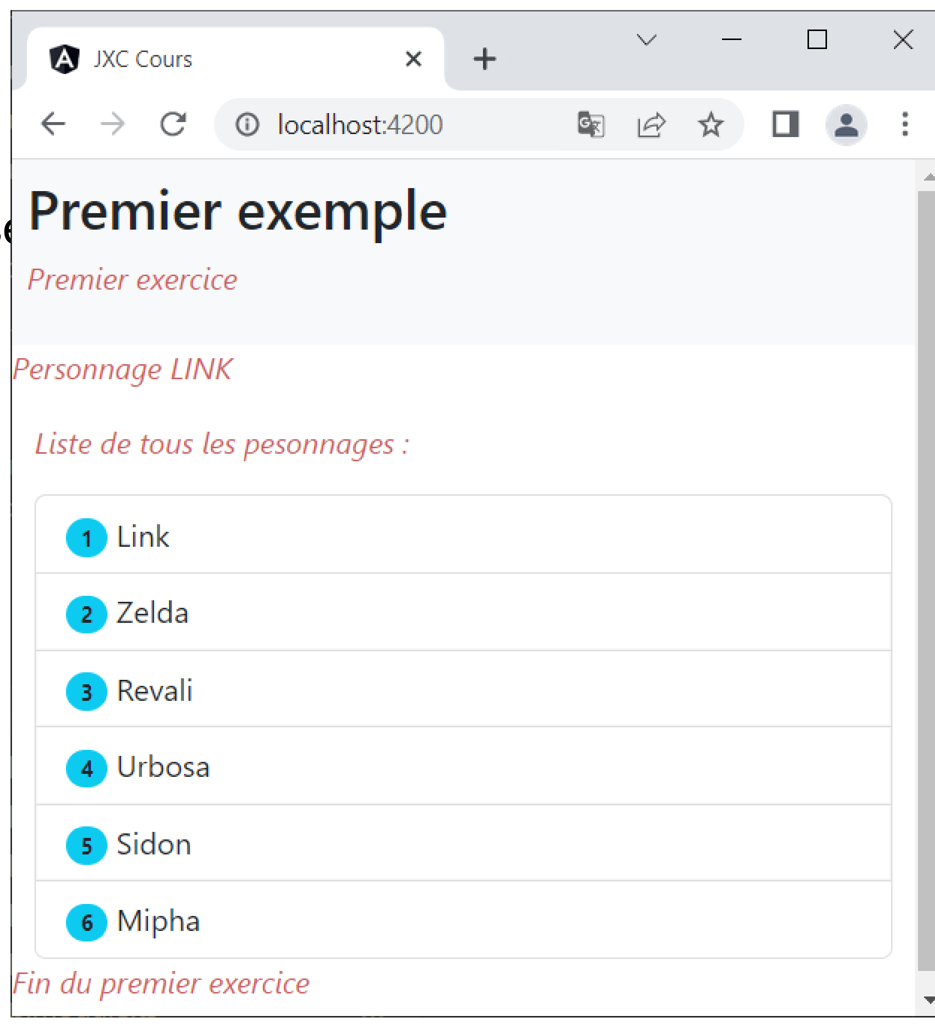
→ Le composant App n’est pas modifié



ANGULAR : VIEW ENCAPSULATION

→ **Exemple :** ajout d’une balise <p> dans

ajout d’un style pour la balise



→ ViewEncapsulation **None** dans Personnages

→ Le style de <p> est celui défini dans Personnages.

→ Le style de <p> dans le composant App est modifié.

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

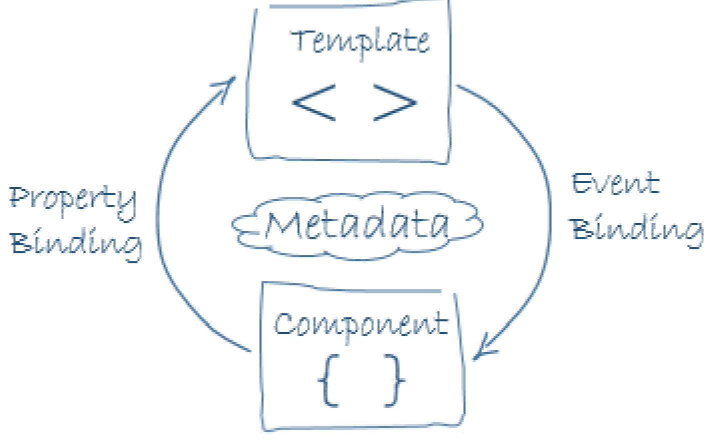
Concept d’Angular

Component / Template

Data Binding

ANGULAR : DATA BINDING

→ Permet de créer une relation entre les données d'un composant et les valeurs correspondantes affichées dans la vue.



Différentes catégories de binding :

→ Du composant à la vue **[ ]**

→ De la vue au composant **( )**

→ Et dans les deux directions - two-way binding **[( )]**

{{value}}
[property] = "value"
(event) = "handler"
[(ng-model)] = "property"
DOM
COMPONENT

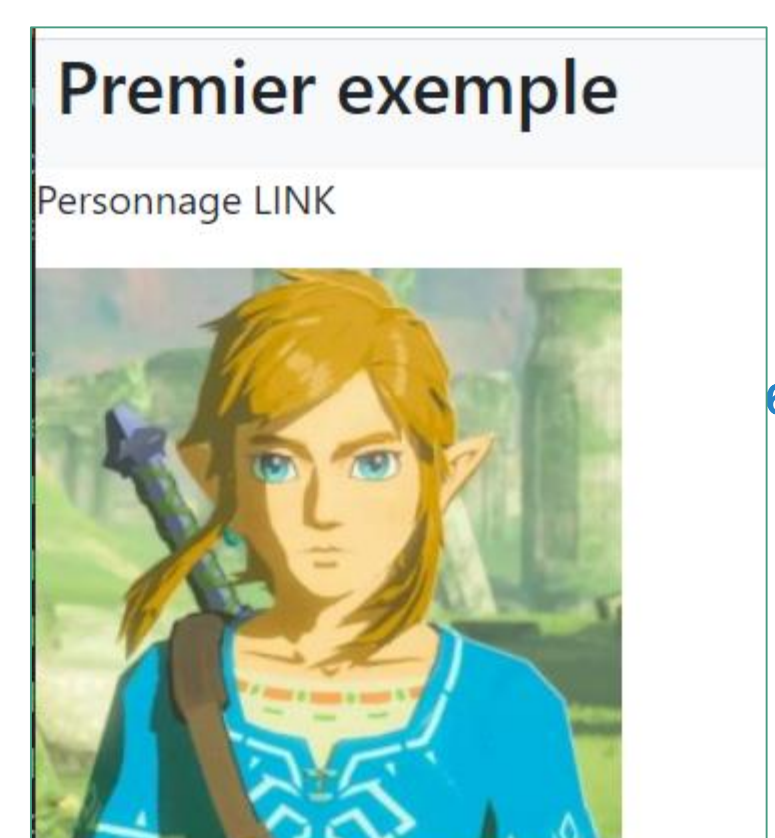
ANGULAR : DATA BINDING

Interpolation :

Une **variable scalaire** est injectée dans le **template**.

→ Si un composant modifie les données, la vue sera automatiquement mise à jour.

```
cours > src > app > personnages > <> personnages.component.html >
1    <p>Personnage {{myHero.name | uppercase}}</p>
2    <img src={{myHero.urlImage}}/>
```



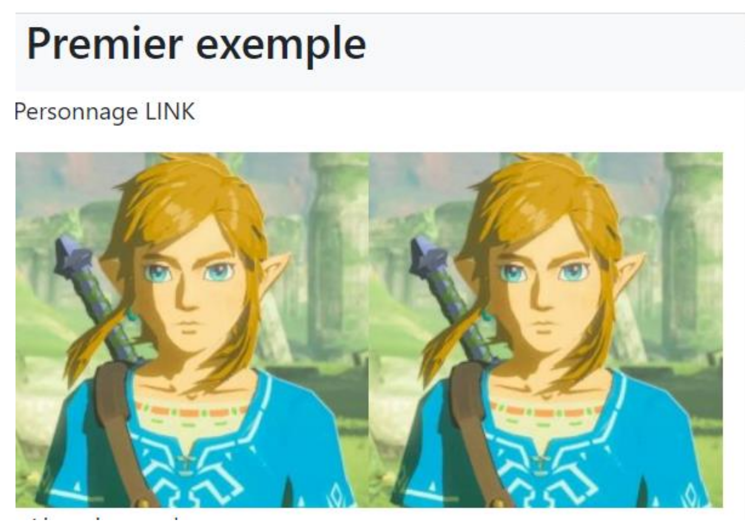
ANGULAR : DATA BINDING

Property binding :

Si la variable à interpoler dans le template est la valeur d’un attribut d’une balise HTML, la gestion de cet attribut peut être délégué à Angular.

→ Attribut HTML encadré par des crochets, devient une directive de l’attribut.

```
rs > jxc-cours > src > app > personnages > <> personnages.component.html >
1    <p>Personnage {{myHero.name | uppercase}}</p>
2    <img src={{myHero.urlImage}}/>
3    <img [src]="myHero.urlImage"/>
```



ANGULAR : DATA BINDING

Property binding : **[target]="expression"**

Le target peut être une propriété d’un élément, d’un composant ou d’une directive.

```
<img [src]="heroImageUrl">
<app-hero-detail [hero]="currentHero"></app-hero-detail>
<div [ngClass]="{'special': isSpecial}"></div>
```

Cette syntaxe utilisé également pour les attributs, classe et style :

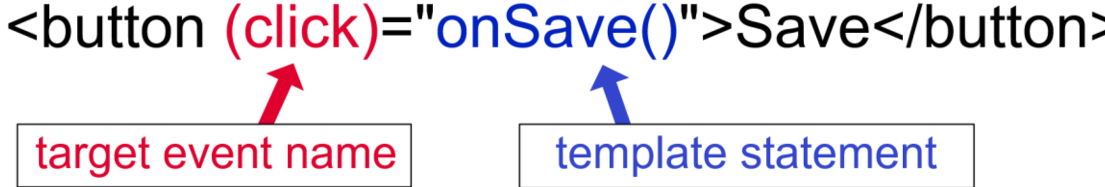
```
<button [attr.aria-label]="help">help</button>
<div [class.special]="isSpecial">Special</div>
<button [style.color]="isSpecial ? 'red' : 'green'"></button>
```

ANGULAR : DATA BINDING

Event binding :

Permet à Angular d’exécuter du code ou des actions lorsqu’un évènement est levé.

→ Clics, mouvements de souris, frappes au clavier, manipulations tactiles, etc.




```
<button (click)="onSave()">Save</button>
<app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>
<div (myClick)="clicked=$event" clickable>click me</div>
```

→ Evènement sur un élément, un composant ou un directive.

ANGULAR : DATA BINDING

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages.

→ Simuler des données d'un back pour les afficher dans le template de Personnages

→ Modification du comportement du composant Personnages (ajout databing).

ANGULAR : EXEMPLE

Modification de la classe TS du composant Personnages

```
11  export class PersonnagesComponent implements OnInit {
12
13    listPersonnages = PERSONNAGES;
14    myHero!: IPersonnage;
15
16    constructor() { }
17    ngOnInit(): void {
18    }
19
20    selectedPersonnage(hero: IPersonnage){
21      this.myHero = hero;
22    }
23  }
```

ANGULAR : EXEMPLE

Modification du template du composant Personnages

CodeCours > jxc-cours > src > app > personnages > personnages.component.html > ...

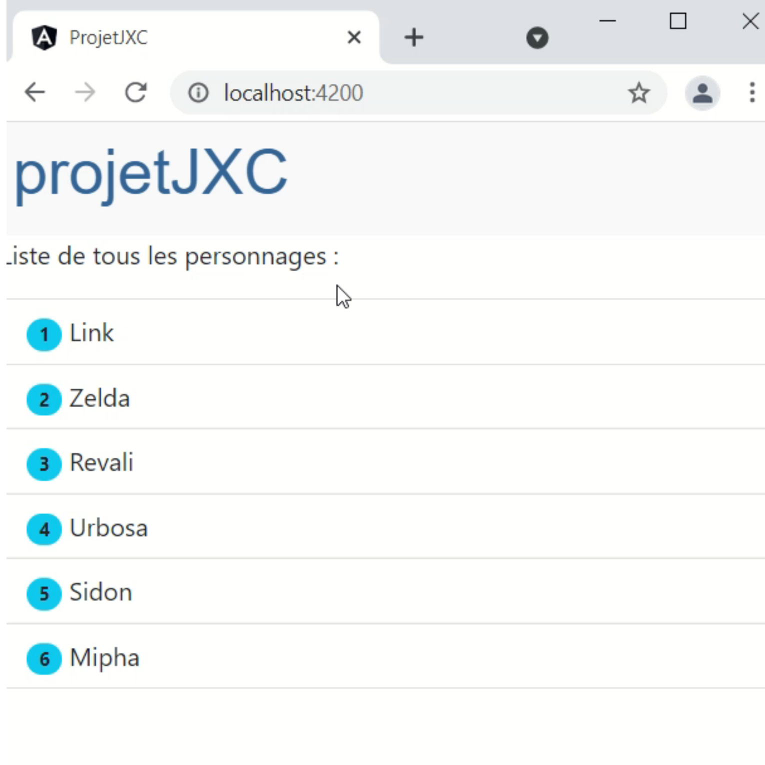
```html
<div class="container">
    <p>Liste de tous les pesonnages :</p>
    <ul class="list-group personnages">
        <li *ngFor="let hero of listPersonnages"
            class="list-group-item"
            [class.selected]="hero===myHero"
            (click)="selectedPersonnage(hero)">
            <span class="badge rounded-pill bg-info text-dark">{{hero.id}}</span>
            {{hero.name}}
        </li>
    </ul>
</div>
<div *ngIf="myHero">
    <p>Personnage {{myHero.name | uppercase}}</p>
    <img [src]="myHero.urlImage"/>
</div>
```

ANGULAR : DATA BINDING

Modification du fichier CSS

projetJXC > src > app > personnages > # personnages.component.css > ...

```css
.personnages li{
    cursor: pointer;
}
.personnages li.selected {
    background-color: rgb(247, 149, 174);
    color: white;
}
```



ANGULAR : DATA BINDING

Event binding :

→ Possibilité d'avoir l'objet **$event** comme paramètre à la méthode.

→ Le type de l'objet **$event** dépend du target (DOM element event)

InputEvent

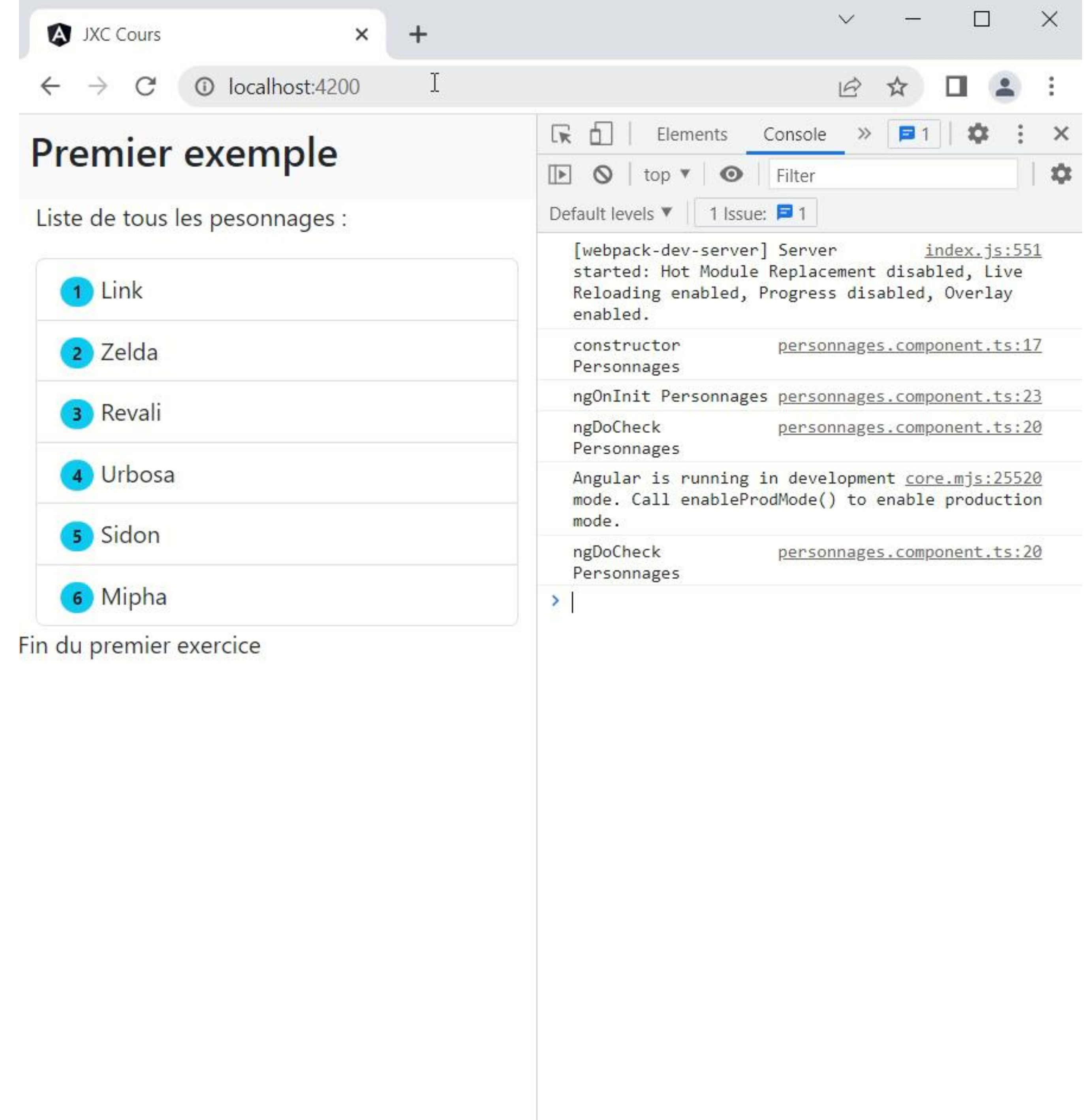
```
<div *ngIf="myHero">
    <p>Personnage {{myHero.name | uppercase}}</p>
    <img [src]="myHero.urlImage"/>
    <input [value]="myHero.name"
        (input)="myHero.name=getValue($event)"/>
</div>
```

```
getValue(event: Event): string {
  console.dir(event);
  return (event.target as HTMLInputElement).value;
}
```

ANGULAR :

Event binding :

→ Résultat :



ANGULAR : DATA BINDING

Two-way binding : **[(target)]="expression"**

Permet de lier une propriété de la classe TypeScript implémentant le composant avec une interface de saisie/sélection du template (input, select, textarea, etc.)

→ Une modification dans le template de la valeur de la zone de saisie met à jour immédiatement la valeur de la propriété dans le classe.

→ La mise à jour de la variable au sein de la propriété est immédiatement répercutée dans le template.

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

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- **Component Interaction**

ANGULAR : INTERACTION ENTRE COMPOSANT

Une application *Angular* est composée de plusieurs composants.

→ De quelle manière les composants peuvent interagir entre eux ?

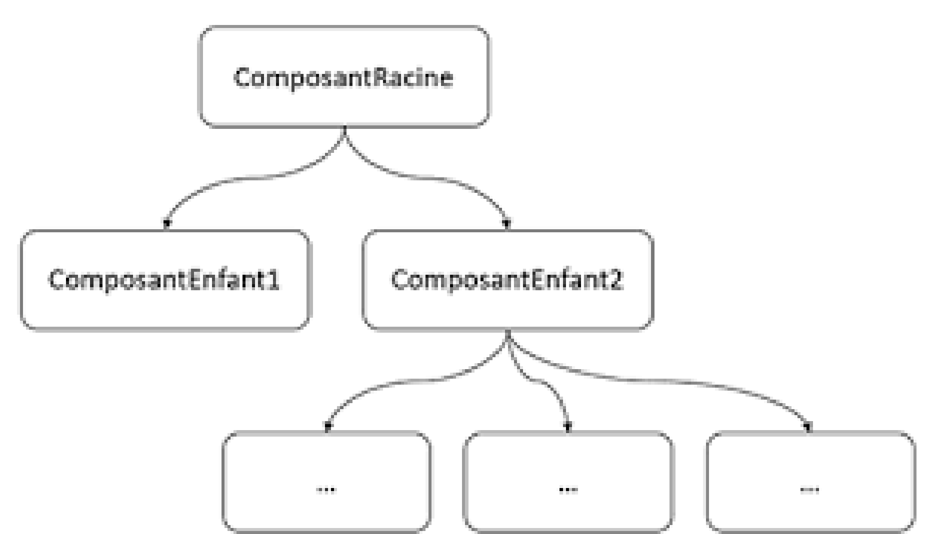
→ **Déclarer les inputs et outputs d'un composant**

→ **EventEmitter**

→ **Getter et Setter**

→ **Variable locale**

→ **Décorateur ViewChild**



ANGULAR : INTERACTION ENTRE COMPOSANT

JXC Cours

localhost:4200

Premier exemple

parent composant

Items:

item1

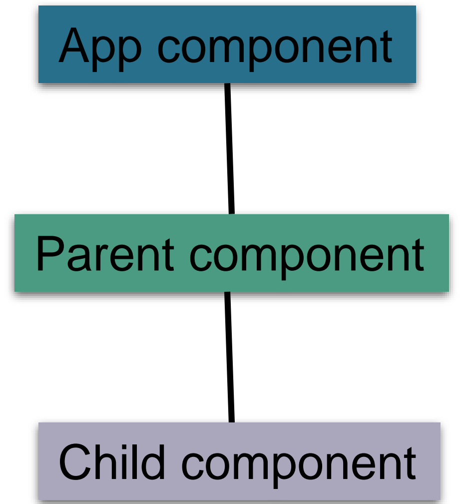
item2

item3

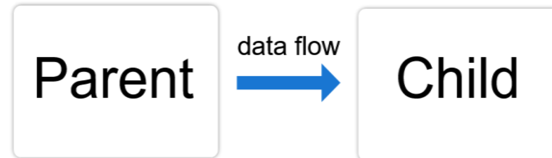
child component

item from parent : item parent ?

Add an item: Add to parent's list



ANGULAR : INTERACTION ENTRE COMPOSANT



@Input :

→ Permet d'identifier une propriété du composant en tant qu'input.

→ Permet au **composant parent** de mettre à jour une donnée du **composant enfant.**

TS du composant enfant :

```
import { Component, Input } from '@angular/core';
export class ItemDetailComponent {
  @Input() item = '';
}
```

Source (propriété du parent)

Template du composant parent :

```
<app-item-detail [item]="currentItem"></app-item-detail>
```

Target (propriété de l'enfant)

ANGULAR : INTERACTION ENTRE COMPOSANT

JXC Cours

localhost:4200

Premier exemple

parent composant

Items:

item1

item2

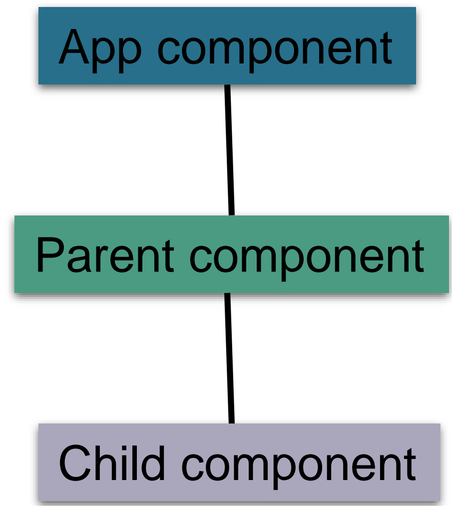
item3

child component

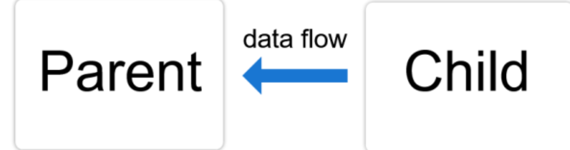
item from parent : item parent

Add an item: Add to parent's list

?



ANGULAR : INTERACTION ENTRE COMPOSANT



@Ouput :

→ Permet de notifier le composant parent qu'un évènement s'est produit au sein du composant enfant.

→ Utilisation de la classe **EventEmitter** afin de notifier le composant parent (évènements personnalisés)

TS du composant enfant :

```
import { Output, EventEmitter } from '@angular/core';
export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

ANGULAR : INTERACTION ENTRE COMPOSANT

Template du composant enfant :

```
<label for="item-input">Add an item:</label>
<input type="text" id="item-input" #newItem>
<button (click)="addNewItem(newItem.value)">Add to parent's list</button>
```


Template du composant parent :

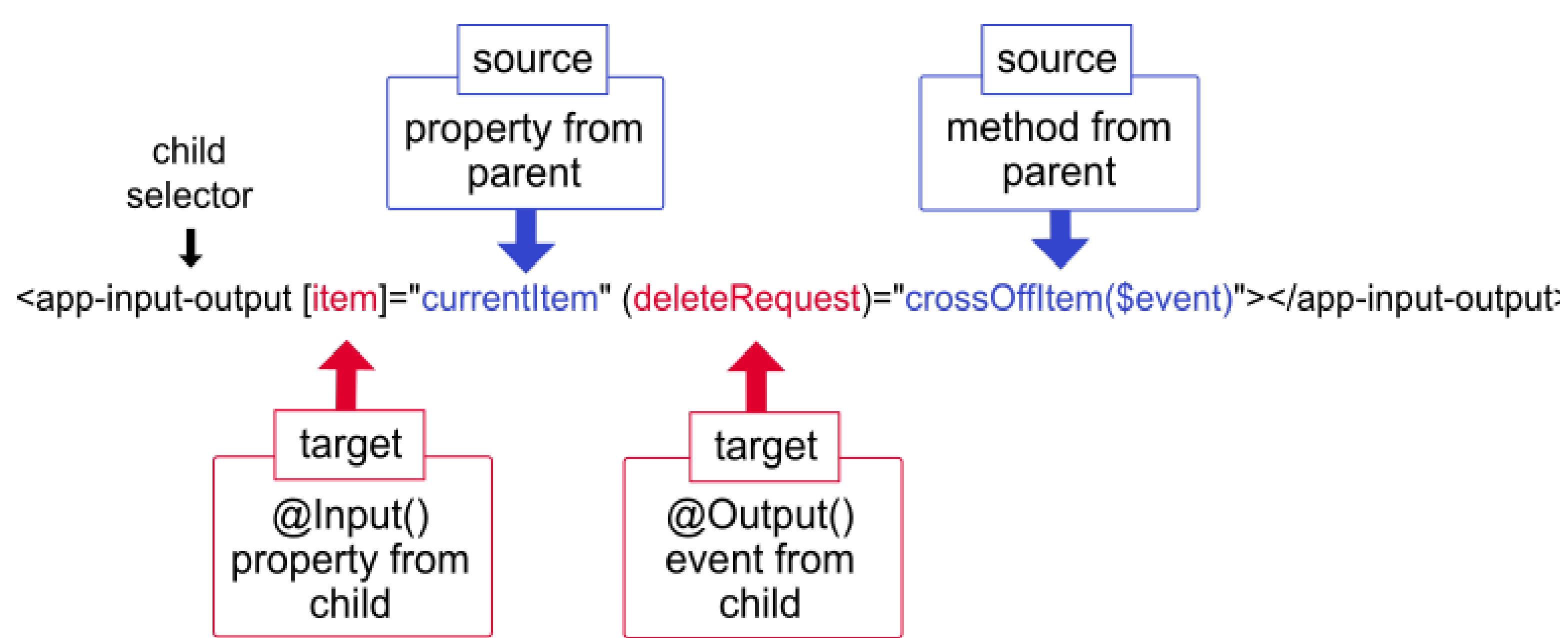
```
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
```

TS du composant parent :

```
export class AppComponent {
  items = ['item1', 'item2', 'item3', 'item4'];

  addItem(newItem: string) {
    this.items.push(newItem);
  }
}
```

ANGULAR : INTERACTION ENTRE COMPOSANT

Possibilité de combiner **@Input** et **@Output** :



ANGULAR : EXEMPLE

→ Création d’un nouveau composant : personnages-detail

→ **Composant parent** : personnages, **composant enfant** : personnages-detail

→ Utilisation du décorateur **@Input** et **@Output**

→ But :

Séparer la liste des personnages et le détail d’un personnage,

ajouter des nouvelles armes à un personnage (ajout d’une donnée arme de type string[ ]).

TS du **composant enfant** (personnages-detail) :

```
CodeCours > jxc-cours > src > app > personnages-detail > TS personnages-detail.component.ts > ...
1   import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2   import { IPersonnage } from '../personnages/IPersonnage';
3
4   @Component({
5     selector: 'app-personnages-detail',
6     templateUrl: './personnages-detail.component.html',
7     styleUrls: ['./personnages-detail.component.css']
8   })
9   export class PersonnagesDetailComponent implements OnInit {
10
11    @Input() hero?: IPersonnage;
12
13    @Output() newArmeEvent = new EventEmitter<string>();
14
15    constructor() { }
16    ngOnInit(): void {
17    }
18
19    changerArme(value: string){
20      this.newArmeEvent.emit(value);
21    }
22  }
```

ANGULAR : EXEMPLE

Template du **composant enfant** (personnages-detail) :

```
CodeCours > jxc-cours > src > app > personnages-detail > <> personnages-detail.component.html > ...
1  <div *ngIf="myHero">
2      <p>Détail :</p>
3      <p>Personnage {{myHero.name | uppercase}}</p>
4      <img [src]="myHero.urlImage"/>
5      <label for="arme-input">Quelle arme ? </label>
6      <input type="text" id="arme-input" #nouvelleArme/>
7      <button (click)="changerArme(nouvelleArme.value)">Modifier</button>
8  </div>
```

TS du **composant parent** (personnages) :

CodeCours > jxc-cours > src > app > personnages > TS personnages.component.ts > ...

```
import { Component, DoCheck, OnInit, ViewEncapsulation } from '@angular/core';
import { PERSONNAGES } from '../mock-personnages';
import { IPersonnage } from './IPersonnage';

@Component({
  selector: 'app-personnages',
  templateUrl: './personnages.component.html',
  styleUrls: ['./personnages.component.css'],
  encapsulation: ViewEncapsulation.Emulated
})
export class PersonnagesComponent implements OnInit {
  listPersonnages = PERSONNAGES;
  myHero!: IPersonnage;
  constructor() { console.log("constructor Personnages"); }
  ngOnInit(): void { console.log("ngOnInit Personnages");  }
  selectedPersonnage(hero: IPersonnage){
    this.myHero = hero;
  }
  modifierArme(newArme: string){
    this.myHero.arme.push(newArme);
  }
}
```

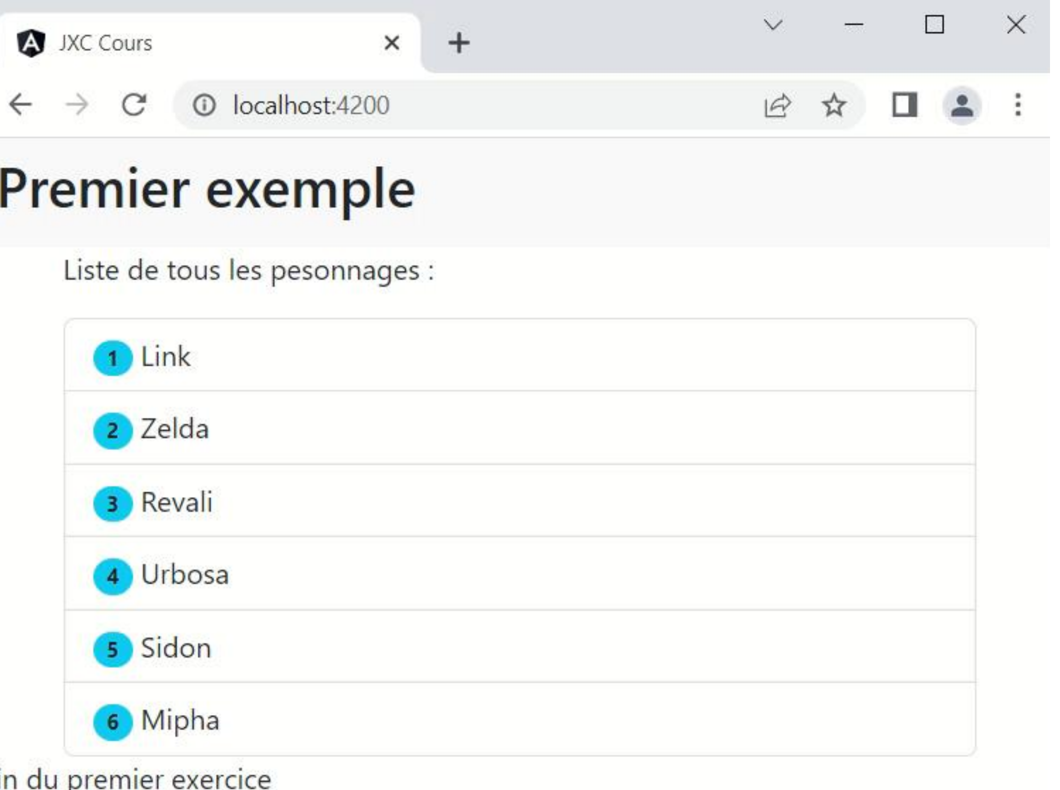
ANGULAR : EXEMPLE

Template du composant parent (personnages) :

```
CodeCours > jxc-cours > src > app > personnages > <> personnages.component.html > ...
1   <div class="container">
2       <p>Liste de tous les pesonnages :</p>
3       <ul class="list-group personnages">
4           <li *ngFor="let hero of listPersonnages"
5               class="list-group-item"
6               [class.selected]="hero===myHero"
7               (click)="selectedPersonnage(hero)">
8               <span class="badge rounded-pill bg-info text-dark">{{hero.id}}</span>
9               {{hero.name}}
10              <span *ngFor="let arme of hero.arme"> {{arme}}</span>
11      </ul>
12  </div>
13
14  <app-personnages-detail [myHero]="myHero" (newArmeEvent)="modifierArme($event)"></app-personnages-detail>
15
```

ANGULAR :

Résultat :



ANGULAR : INTERACTION ENTRE COMPOSANT

setter

→ Possibilité d'intercepter un changement de valeur grâce à un setter


```
export class NameChildComponent {
  @Input()
  get name(): string { return this._name; }
  set name(name: string) {
    this._name = name;
  }
  private _name = '';
}
```

ANGULAR : EXEMPLE

→ Template du **composant parent**

Two-way binding

src > app > test-get-set-parent > <> test-get-set-parent.component.html > ...

```html
1  <div>
2      <input [(ngModel)]="text" placeholder="Texte"/>
3      <hr/>
4      <app-test-get-set  [name]="text"></app-test-get-set>
5  </div>
```

One-way binding : property binding

projetJXC

Parent

Texte

Enfant

Mon nom :

Historique :

-

ANGULAR : EXEMPLE

→ TS du **composant parent** :

src > app > test-get-set-parent > TS test-get-set-parent.component.ts > ...

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-test-get-set-parent',
  templateUrl: './test-get-set-parent.component.html',
  styleUrls: ['./test-get-set-parent.component.css']
})
export class TestGetSetParentComponent implements OnInit {

  text: string = '';

  constructor() { }
  ngOnInit(): void {
  }
}
```

ANGULAR :

→ TS du **composant enfant**

src > app > test-get-set > TS test-get-set.component.ts > ...

```
import { Component, OnInit, Input } from '@angular/core';
@Component({
  selector: 'app-test-get-set',
  templateUrl: './test-get-set.component.html',
  styleUrls: ['./test-get-set.component.css']
})
export class TestGetSetComponent implements OnInit {
  historyName: string[] = [];
  private _name: string = '';
  get name(): string {
    return this._name;
  }
  @Input()
  set name(value: string){
    this._name = value;
    this.historyName.push(value);
  }
  constructor() { }
  ngOnInit(): void {
  }
}
```

ANGULAR : EXEMPLE

→ Template du **composant enfant**

```
src > app > test-get-set > <> test-get-set.component.html > ...
1    <p>Mon nom : {{ name }}</p>
2    <div>
3        <p>Historique :</p>
4        <ul>
5            <li *ngFor="let oldName of historyName">
6                {{oldName}}
7            </li>
8        </ul>
9    </div>
```

ANGULAR : INTERACTION ENTRE COMPOSANT

<u>variable locale</u> (template référence variable) :

→ Permet d’utiliser une donnée déclarée dans un **template** à un autre endroit de ce **template**.

→ Peut se référer à :

Un élément du DOM dans le template (e.g. input),

Un composant,

Une directive

#templateVariable

ANGULAR : INTERACTION ENTRE COMPOSANT

→ **Composant enfant :**

src > app > testlocalvar-child > TS testlocalvar-child.component.ts > ...

```
8   export class TestlocalvarChildComponent {
9     text: string = '';
10    showMessage() {
11      alert(this.text);
12    }
13  }
```

src > app > testlocalvar-child > <> testlocalvar-child.component.html > ...

```
1   <div>
2       <p class="text-danger fw-bold">Partie enfant</p>
3       <input [(ngModel)]="text" placeholder="Texte de l'enfant"/>
4       <p>Texte de l'enfant : {{text}}</p>
5   </div>
```

ANGULAR : INTERACTION ENTRE COMPOSANT

→ **Composant parent :**

src > app > testlocalvar-parent > TS testlocalvar-parent.component.ts > ...

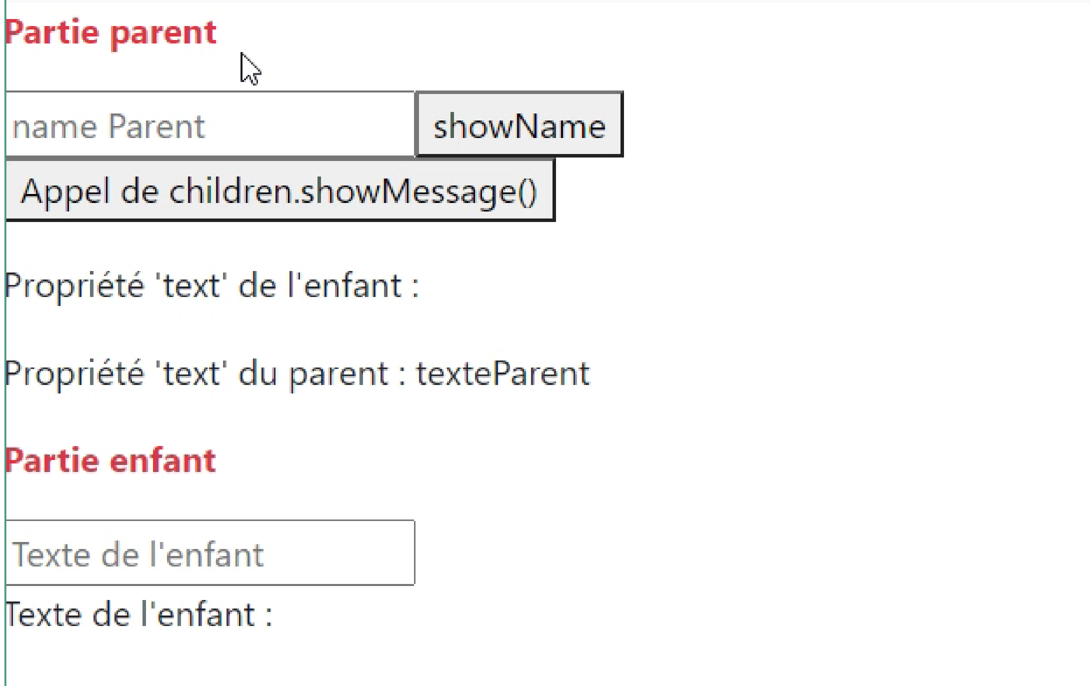
```
8   export class TestlocalvarParentComponent {
9     text: string = 'texteParent';
10    showName(value: string){
11      alert(value);
12    }
13  }
```


src > app > testlocalvar-parent > <> testlocalvar-parent.component.html > ...

```
1  <div>
2      <p class="text-danger fw-bold">Partie parent</p>
3      <input #name placeholder="name Parent" />
4      <button (click)="showName(name.value)">showName</button>
5      <button (click)="children.showMessage()">Appel de children.showMessage()</button>
6      <p>Propriété 'text' de l'enfant : {{children.text}}</p>
7      <p>Propriété 'text' du parent : {{text}}</p>
8      <app-testlocalvar-child #children></app-testlocalvar-child>
9  </div>
```

ANGULAR : INTERACTION ENTRE COMPOSANT

→ Résultat



ANGULAR : INTERACTION ENTRE COMPOSANT

@ViewChild :

Permet de récupérer les données du premier composant fils à partir d'un composant parent.

```
export class PereComponent implements OnInit, AfterViewInit {
  @ViewChild(FilsComponent)
  private fils!: FilsComponent;
  constructor(){}
  ngAfterViewInit() {};
  ngOnInit(){};
}
```


@ViewChildren :

Possibilité à un composant parent de récupérer les données de ses composants enfants (QueryList).

ANGULAR : EXEMPLE

→ Modification de l’exemple précédent (même résultat)

```
export class TestlocalvarParentComponent implements AfterViewInit{
  @ViewChild (TestlocalvarChildComponent, {static: false})
  childComp !: TestlocalvarChildComponent;
  text: string = 'texteParent';
  showName(value: string){
    alert(value);
  }
  callChildren(){
    this.childComp.showMessage();
  }
  ngAfterViewInit(){
    //composant disponible ici
  }
}
```

ANGULAR : EXEMPLE

→ Modification de l’exemple précédent (même résultat)

```
src > app > testlocalvar-parent > <> testlocalvar-parent.component.html > ...
1   <div>
2       <p class="text-danger fw-bold">Partie parent</p>
3       <input #name placeholder="name Parent" />
4       <button (click)="showName(name.value)">showName</button>
5       <p><button (click)="callChildren()">Appel de children.showMessage()</button></p>
6       <app-testlocalvar-child></app-testlocalvar-child>
7   </div>
```

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- **Service**

ANGULAR : SERVICE

→ Un composant doit se focaliser sur la représentation des données (user experience). Il ne doit pas se soucier des mécanismes et transformations mis en place pour récupérer ces données.

→ **Délégation du traitement des données au service** (fetching data, validation d'une saisie utilisateur, système de log).

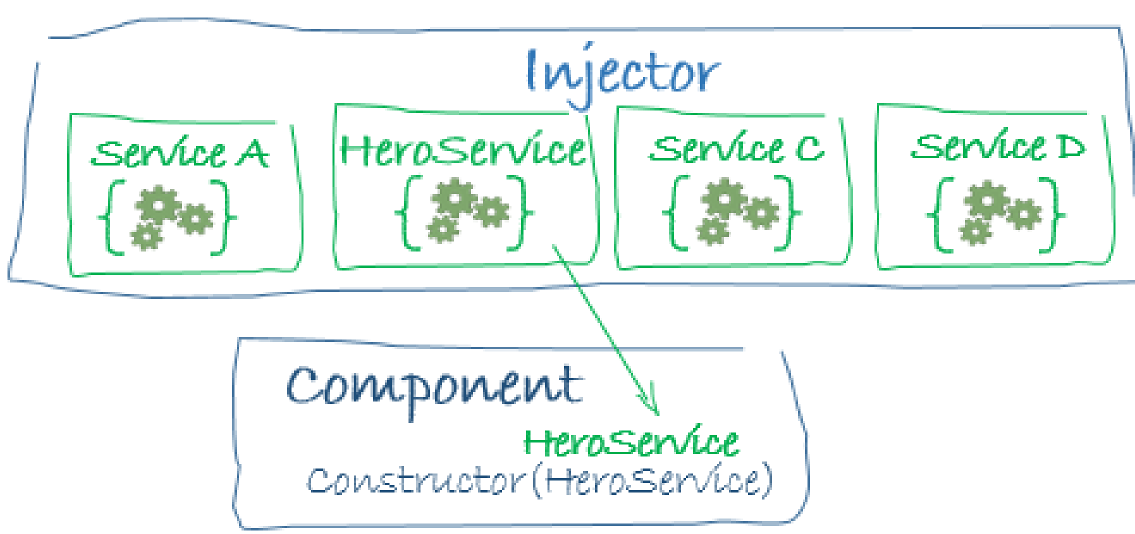
→ Un service permet de partager facilement des informations entre classes qui n'ont pas de lien.

→ **Permet d'augmenter la modularité et la réutilisation des composants.**

ANGULAR : INJECTION DE DÉPENDANCE

→ Les dépendances sont des services ou des objets dont une classe à besoin.

→ Les injections de dépendances (DI) est un design pattern.



ANGULAR : SERVICE

Déclaration d’un service (à partir de Angular 6)

Décorateur : Angular peut utiliser cette classe dans le DI

```
src > app > TS personnage.service.ts > ...
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    providedIn: 'root'
5  })
6  export class PersonnageService {
7
8    constructor() { }
9  }
```

Visible dans toute l’application.

Ajout au provider.

ANGULAR : SERVICE

Déclaration d'un service (avant Angular 6)

```
import { Injectable } from '@angular/core';
@Injectable()
export class PersonneService {
  constructor() { }
}
```

ANGULAR : INJECTION DE DÉPENDANCE

Injection d'un service

→ Pas d'utilisation du **new**

→ Dans le constructeur du composant ayant besoin du service :

```
constructor(private persoService:PersonnageService) { }
```

Singleton de *PersonnageService*

ANGULAR : INJECTION DE DÉPENDANCE

Autre manière possible (versions Angular inférieures à 6)

→ Utilisation de **providers** (metadata) pour spécifier les services dont le composant à besoin.

Dans un composant :

```
@Component({
  selector: 'app-personnages',
  templateUrl: './personnages.component.html',
  styleUrls: ['./personnages.component.css'],
  providers:[PersonnageService]
})
export class PersonnagesComponent implements OnInit {
```

Dans un module :

```
@NgModule({
  declarations: [
    AppComponent,
```

```
  ],
  providers: [LoggerService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

ANGULAR : INJECTION DE DÉPENDANCE

Exemple pour la partie Angular basé sur Zelda :

→ Utilisation du composant racine du projet Angular.

→ Nouveau composant Personnages.

→ Simuler des données d'un back pour les afficher dans le template de Personnages

→ Modification du comportement du composant Personnages (ajout databing).

→ Création d'un nouveau composant : personnages-detail

→ Création de services pour récupérer les données des Personnages et log.

ANGULAR : EXEMPLE

→ Récupération du **mock des personnages dans un service** et **utilisation d'un système de log** :

Service Logger :

CodeCours > jxc-cours > src > app > services > TS logger.service.ts > ...

```typescript
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class LoggerService {
  logs: string[] = []; //capture logs

  constructor() { }

  log(message: string){
    this.logs.push(message);
    console.log(message);
  }
}
```

ANGULAR : EXEMPLE

Service Personnage :

CodeCours > jxc-cours > src > app > services > TS personnage.service.ts > ...

```
import { Injectable } from '@angular/core';
import { PERSONNAGES } from '../mock-personnages';
import { IPersonnage } from '../personnages/IPersonnage';
import { LoggerService } from './logger.service';

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

  constructor(private loggerService: LoggerService) { }
  getPersonnages(): IPersonnage[]{
    this.loggerService.log('Getting personnages...');
    return PERSONNAGES;
  }
}
```

Composant

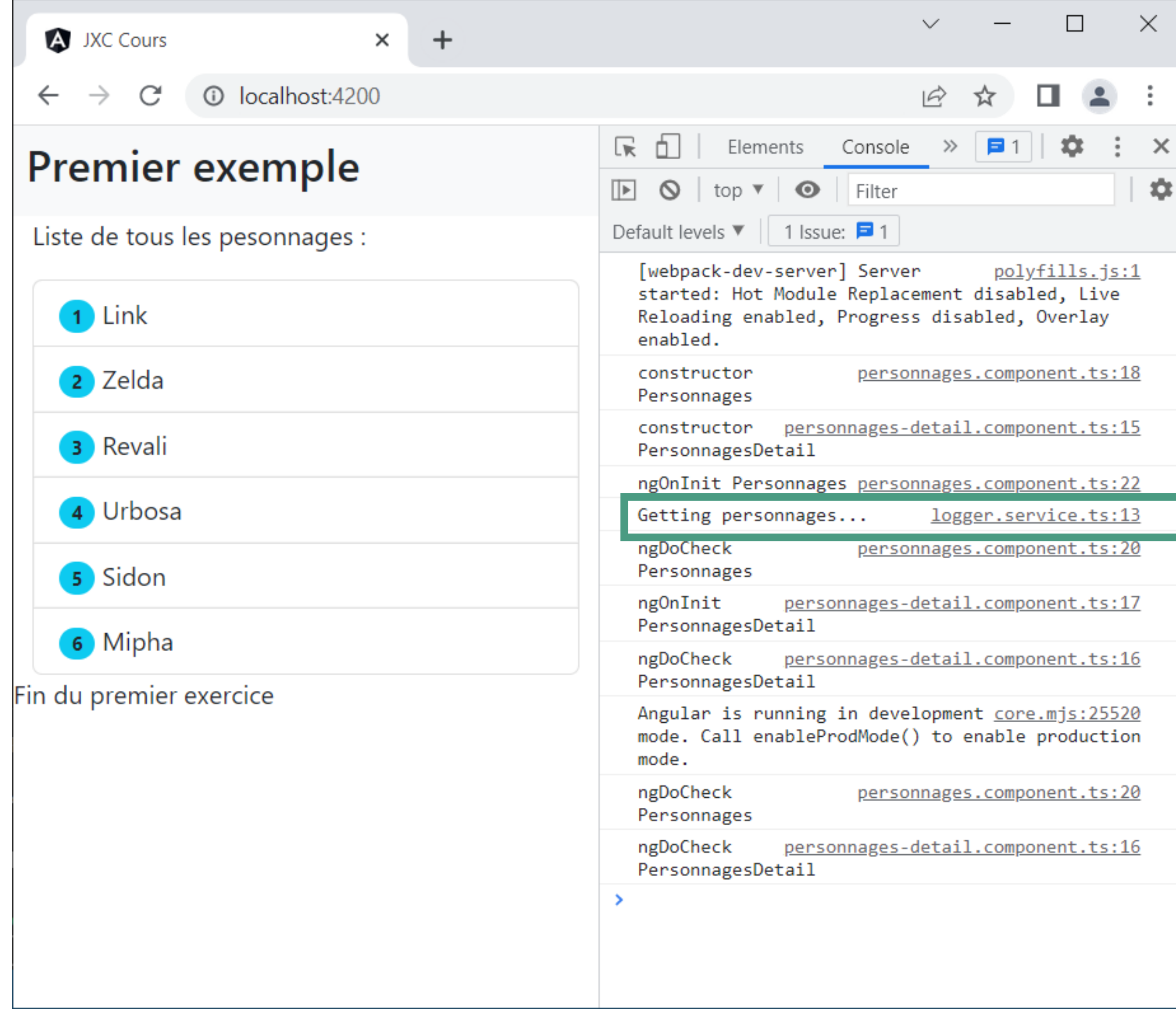
Personnages :

CodeCours > jxc-cours > src > app > personnages > TS personnages.component.ts > ...

```
import { Component, DoCheck, OnInit, ViewEncapsulation } from '@angular/core';
//import { PERSONNAGES } from '../mock-personnages';
import { PersonnageService } from '../services/personnage.service';
import { IPersonnage } from './IPersonnage';

@Component({
  selector: 'app-personnages',
  templateUrl: './personnages.component.html',
  styleUrls: ['./personnages.component.css'],
  encapsulation: ViewEncapsulation.Emulated
})
export class PersonnagesComponent implements OnInit, DoCheck {
  listPersonnages!: IPersonnage[];  // = PERSONNAGES
  myHero!: IPersonnage;
  constructor(private personnageService: PersonnageService) { }
  ngDoCheck(): void { console.log("ngDoCheck Personnages"); }
  ngOnInit(): void {
    console.log("ngOnInit Personnages");
    this.getHeros();
  }
  getHeros(): void {
    this.listPersonnages = this.personnageService.getPersonnages();
  }
  selectedPersonnage(hero: IPersonnage){
```

ANGULAR : EXEMPLE

→ Résultat :



FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d’Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- **Observable (RXJS)**

ANGULAR : RXJS

Dans un réelle application : besoin de services asynchrones.

→ Utilisation de callback, de **Promise** ou de **Observable** (bibliothèque **RxJS**).

RxJS : Reactive extensions for javascript

→ combine Observer pattern et Iterator pattern.

- Observable, Observer, Subscription
- Operators
- Subject

ANGULAR : OBSERVABLE

Observable :

→ Objet permettant un échange d’information (stream).

→ Utilisé lors d’évènements, par la module HttpClient (get() retourne un Observable), etc.

→ Méthode *subscribe()* : abonne un traitement à l’observable.

→ Méthode *unsubscribe()* : désabonne un traitement à l’observable.

Un **observable** va émettre des évènements qui seront interceptés par les **observateurs**.

ANGULAR : OBSERVABLE

Méthode *subscribe()* : un seul paramètre (Observer)

Un Observer va « surveiller » l'Obervable et recevoir les valeurs émises.

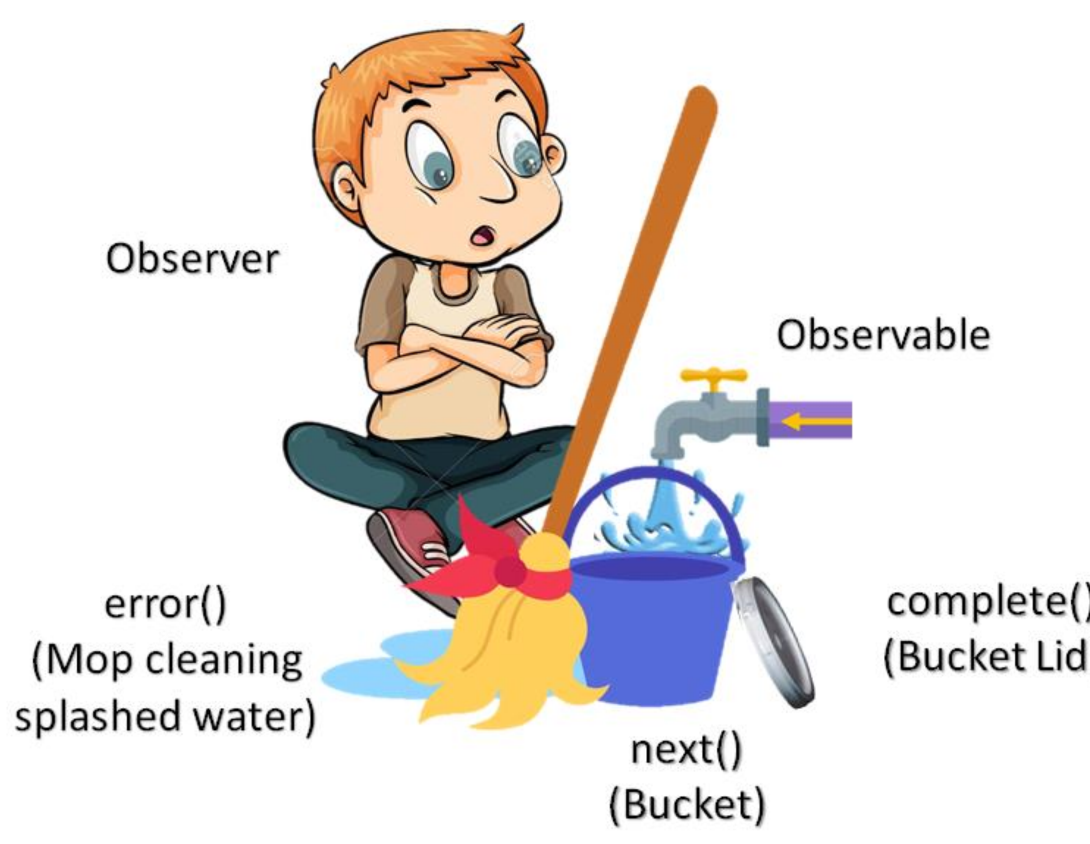
→ **next** : se déclenche à chaque fois que l'Observable émet de nouvelles données (données en tant qu'argument)

→ **error** : se déclenche si l'Observable émet une erreur (erreur en tant qu'argument)

→ **complete** : se déclenche si l'Observable s'achève (aucun argument)

```
monObservable$.subscribe({
  next: (v) => console.log(v),
  error: (e) => console.error(e),
  complete: () => console.info('complete')
});
```

ANGULAR : OBSERVABLE



ANGULAR : OBSERVABLE

→ Plusieurs **opérateurs** de transformation disponibles :

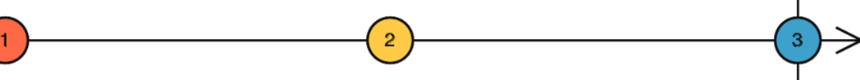
AREA	OPERATORS
Creation	`from`, `fromEvent`, `of`
Combination	`combineLatest`, `concat`, `merge`, `startWith` , `withLatestFrom`, `zip`
Filtering	`debounceTime`, `distinctUntilChanged`, `filter`, `take`, `takeUntil`
Transformation	`bufferTime`, `concatMap`, `map`, `mergeMap`, `scan`, `switchMap`
Utility	`tap`
Multicasting	`share`

ANGULAR : OBSERVABLE

→ Plusieurs manières de **créer** un **Observable**.

```
let observableOf$ = of(1, 2, 3);
//emet chaque valeur/objet passé en paramètre
//output: 1, 2, 3
let observableFrom$ = from([1, 2, 3]);
//utiliser pour des array, promise
//emet le tableau comme une séquence de valeur
//output: 1, 2, 3
let observableInterval$ = interval(3000);
//emet toutes les 3 sec des valeurs
//output: 0, 1, 2, 3, 4 ....
let observableTimer$ = timer(1000,2000);
//commence à emettre la première valeur après 1 sec et emet toutes les 2 sec
//output: 0, 1, 2, 3, 4 ....
let observableEvent$ = fromEvent<MouseEvent>(document, 'mousemove');
//emet lorsque la souris est déplacée
//output: MouseEvent
```

of(1, 2, 3)



from([10, 20, 30])

10 20 30

223

interval(1000)

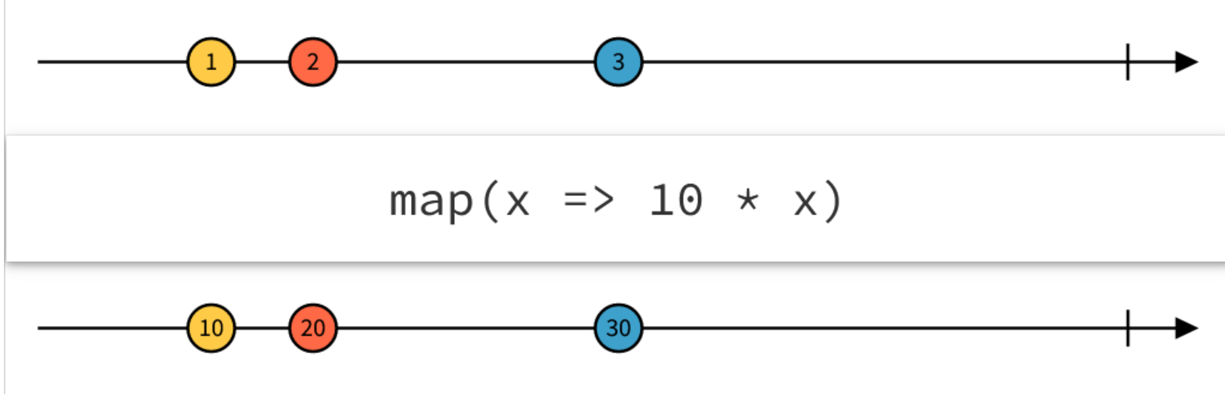
0 1 2 3 4 5

fromEvent(element, 'click')

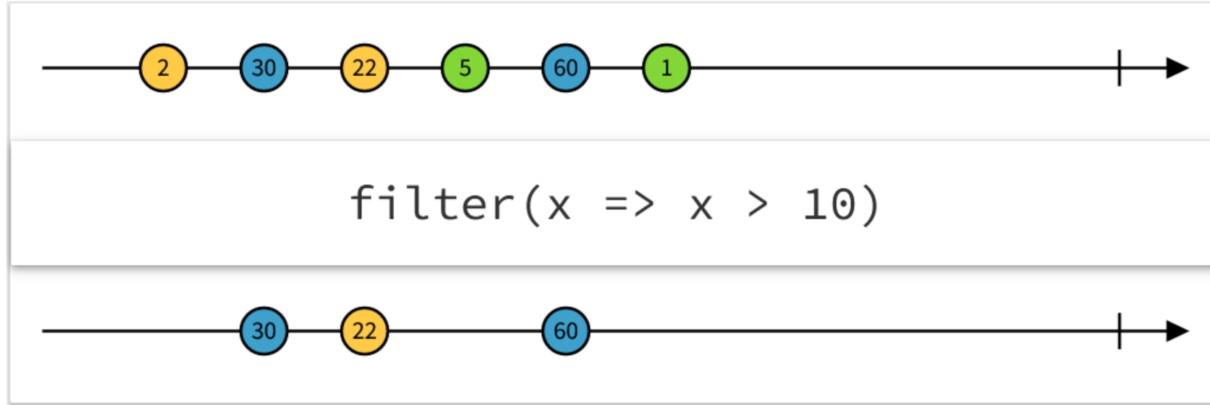
ev ev

ANGULAR : OBSERVABLE

→ **Transformation**

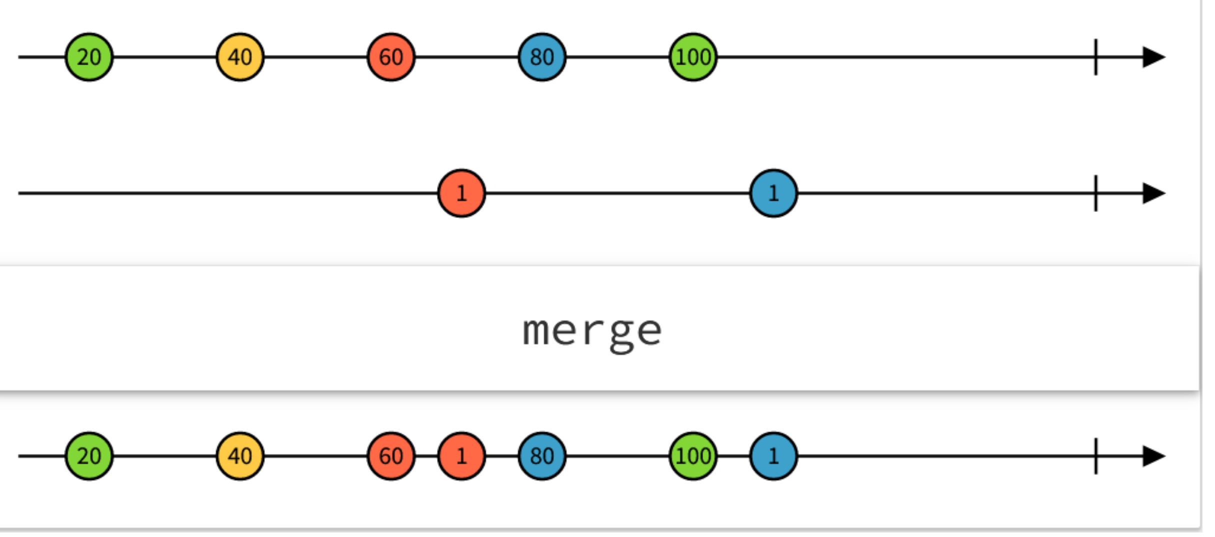


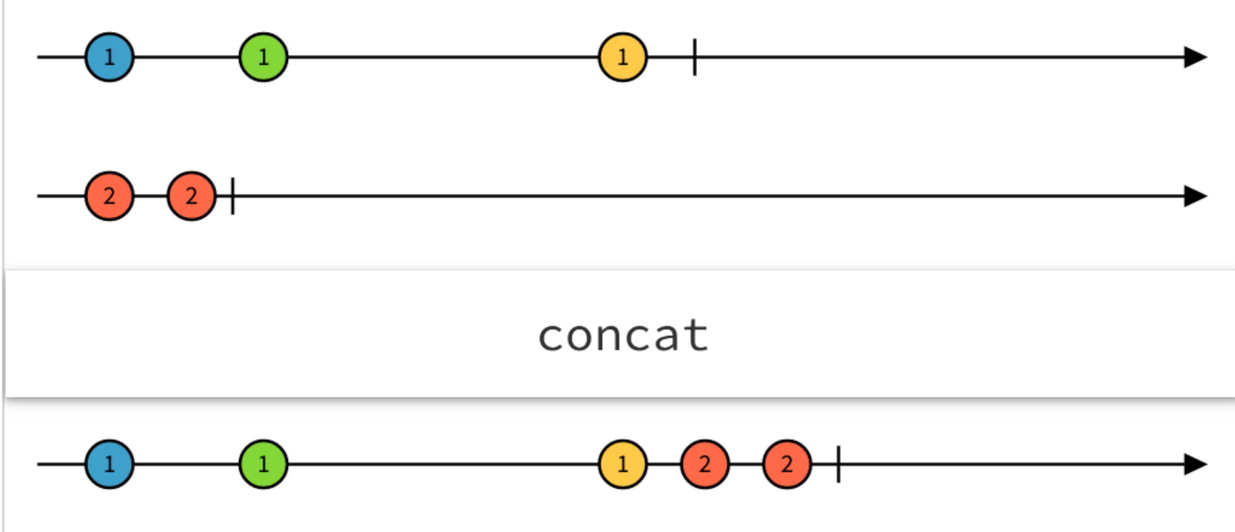
→ **Filtre**



ANGULAR : OBSERVABLE

→ **Combinaison**





ANGULAR : EXEMPLE

→ Modification de l’exemple précédent utilisant un **Observable** (asynchrone)

Service Personnage:

CodeCours > jxc-cours > src > app > services > TS personnage.service.ts > ...

```
1   import { Injectable } from '@angular/core';
2   import { Observable, of } from 'rxjs';
```

```
7   @Injectable({
8     providedIn: 'root'
9   })
10  export class PersonnageService {
11
12    constructor(private loggerService: LoggerService) { }
13
14    getPersonnages(): Observable<IPersonnage[]>{
15      this.loggerService.log('Getting personnages...');
16      return of(PERSONNAGES);
17    }
18  }
```

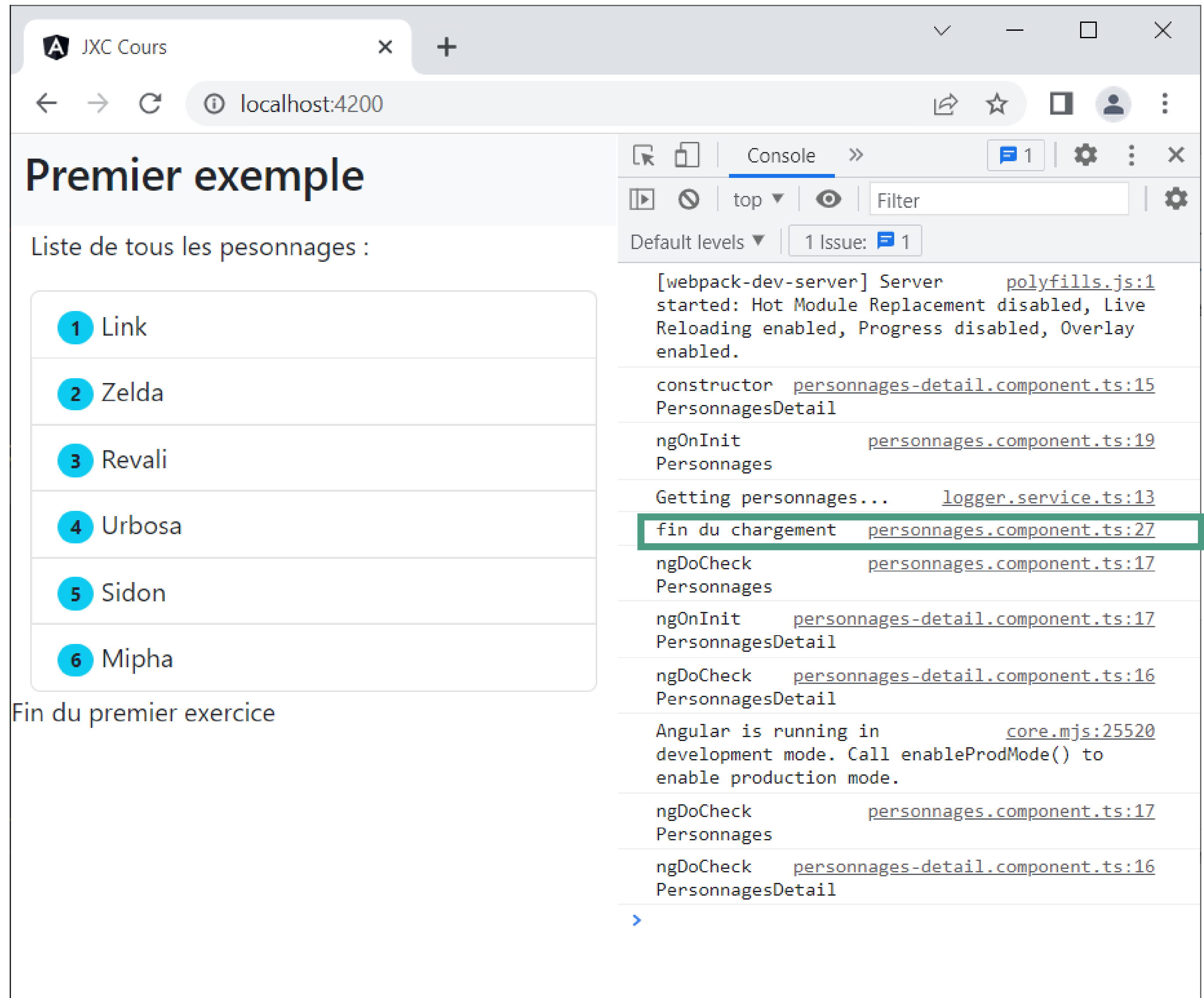
ANGULAR : EXEMPLE

composant Personnage:

```
13  export class PersonnagesComponent implements OnInit, DoCheck {
14
15    listPersonnages!: IPersonnage[];
16    myHero!: IPersonnage;
17
18    constructor(private personnageService: PersonnageService) { }
19    ngDoCheck(): void { console.log("ngDoCheck Personnages"); }
20    ngOnInit(): void {
21      console.log("ngOnInit Personnages");
22      this.getHeros();
23    }
24
25    getHeros(): void {
26      //this.listPersonnages = this.personnageService.getPersonnages();
27      this.personnageService.getPersonnages().subscribe({
28        next: personnages => this.listPersonnages = personnages,
29        error: error => console.error(error),
30        complete: () => console.info('fin du chargement')
31      });
32    }
```

ANGULAR : EXEMPLE

→ **Résultat identique :**



ANGULAR : SUBJECT

Subject

→ Type spécial d'Observable qui permet de partager les valeurs à plusieurs Observable.

→ Un **Subject** est un **Observable** (possibilité de souscrire à un Subject).

→ Un **Subject** est un **Observer** (object avec les callback next, error, complete).

→ **Plusieurs types :** subject, behabiorSubject, replaySubject, asyncSubject

ANGULAR : SUBJECT

```
const subject = new Subject<number>();
subject.subscribe({
next: (val) => console.log( `observerA: ${ v } ` )
});
subject.subscribe({
next: (val) => console.log( `observerB: ${ v } ` )
});
const observable = from([ 1 , 2 , 3 ]);
observable.subscribe(subject);
```

```
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
// observerA: 3
// observerB: 3
```

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d’Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Observable (RxJS)

Rooting

ANGULAR : ROUTING

→ Dans une **SinglePage Application**, le contenu de la page est modifié au fur et à mesure (affichage de différents composants).

→ Notion de **navigation importante** pour les utilisateur (e.g. mimer le mécanisme de copier l'url, réaliser un retour en arrière dans le navigateur)

→ Solution : **créer des routes** afin de définir comment l'utilisateur navigue d'une partie de l'application à une autre.

ANGULAR : ROUTING

→ Mise en place d'un **routeur** : un module situé au niveau le plus haut de l'application dédié à la gestion des routes.

→ Nom du module : **AppRoutingModule**.


```
src > app > TS app-routing.module.ts > ...
1   import { NgModule } from '@angular/core';
2   import { RouterModule, Routes } from '@angular/router';
3
4   const routes: Routes = [];
5
6   @NgModule({
7     imports: [RouterModule.forRoot(routes)],
8     exports: [RouterModule]
9   })
10  export class AppRoutingModule { }
```

Table de routes

Importation du module `Router` (au niveau de la racine de l'application

Module disponible dans toute l'application

ANGULAR : ROUTING

→ Mise en place d'un **routeur** : un module situé au niveau le plus haut de l'application dédié à la gestion des routes.

→ Nom du module : **AppRoutingModule**.

src > app > TS app-routing.module.ts > ...

```
1   import { NgModule } from '@angular/core';
2   import { RouterModule, Routes } from '@angular/router';
3
4   const routes: Routes = [];
5
6   @NgModule({
7     imports: [RouterModule.forRoot(routes)],
8     exports: [RouterModule]
9   })
10  export class AppRoutingModule { }
```

{enableTracing: true}

Importation du module `Router` (au niveau de la racine de l'application

Permet de garder une trace de la recherche d'un chemin (*debug*)

ANGULAR : ROUTING

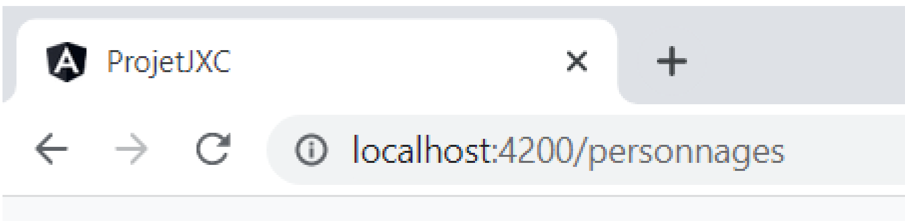
Forme basique d’une route :

→ **Path** : chaîne de caractère pour l’URL

→ **Component** : le composant associé à l’URL


```
const routes: Routes = [
  {path: 'personnages', component: PersonnagesComponent}
];
```

→ Lorsque l’URL est demandée, le module de routage effectue le rendu du composant associé.



ANGULAR : ROUTING

Attention à l'ordre des routes (utilisation de la première route correspondante)

→ Possibilité d'effectuer une redirection :

```
const routes: Routes = [
  {path: 'personnages', component: PersonnagesComponent},
  {path: '', redirectTo: 'personnages', pathMatch: 'full'}
];
```

→ Possibilité d'utiliser des URL dynamiques :

```
{path: 'personnage/:name', component: PersonnagesDetailComponent}
```

localhost:4200/personnage/Mipha

A N G U L A R : R O U T I N G

→ Possibilité de rediriger l'utilisateur lorsqu'il rentre une URL qui n'existe pas (*wildcart route*)

→ Création d'une page 404 (nouveau composant)


```
{path: '**', component: PageNotFoundComponent}
```

→ A ajouter en dernier dans la table de routage !

ANGULAR : ROUTING

Comment intégrer les routes ?

Directive **`RouterOutlet`**

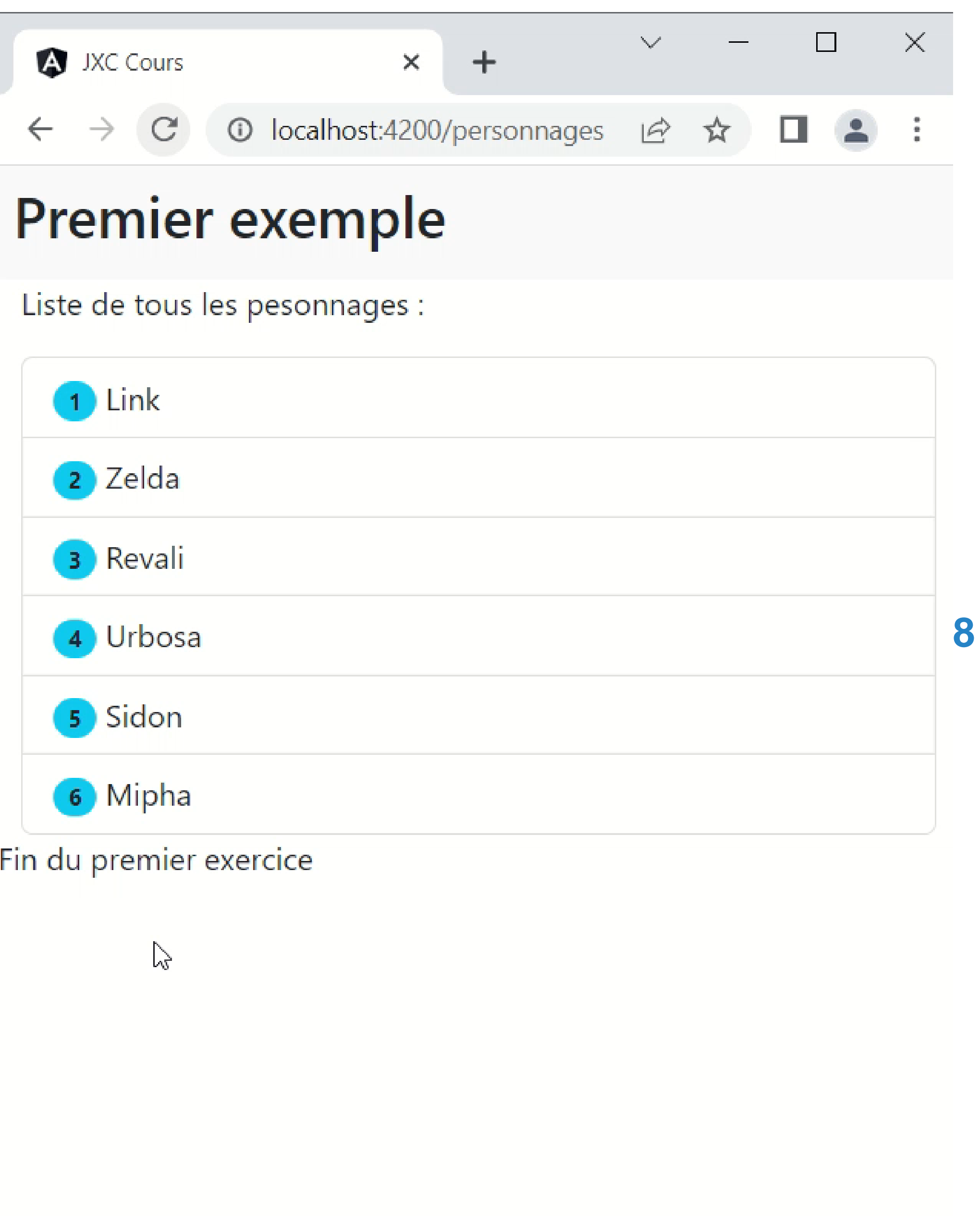
→ Utilisée comme un composant

→ Permet de préciser au module de routage où il doit faire le rendu des composants associés aux routes.

src > app > app.component.html > ...

```
1  <header class="container-fluid bg-light p-2">
2    <h1>{{title}}</h1>
3  </header>
4  <!-- <app-personnages></app-personnages> -->
5  <router-outlet></router-outlet>
```

Le composant chargé sera affiché ici



ANGULAR : ROUTING

Comment naviguer dans l'application ?

1. A travers des liens : **`RouterLink`**

2. Directement dans le code du composant

ANGULAR : ROUTING

Problème : l'élément sera en gras pour toutes les pages visitées

Comment naviguer dans l'application ?

1. A travers des liens : **RouterLink**

```
.active {
    font-weight: bold;
}
```

Route active : on peut lui définir un style

→ Cas pour une **URL non dynamique** :

```
<nav>
    <ul>
      <li><a routerLink="/first-component" routerLinkActive="active">First Component</a></li>
      <li><a routerLink="/second-component" routerLinkActive="active">Second Component</a></li>
    </ul>
</nav>
```

→ Possibilité de créer un composant menu

ANGULAR : ROUTING

Comment naviguer dans l'application ?

1. A travers des liens : **RouterLink**

La classe est uniquement ajoutée lorsque la route correspond exactement à la valeur de *routerLink*

```
[routerLinkActiveOptions] = "{exact: true}"
```


→ Cas pour une **URL non dynamique** :

```
<nav>
    <ul>
      <li><a routerLink="/first-component" routerLinkActive="active">First Component</a></li>
      <li><a routerLink="/second-component" routerLinkActive="active">Second Component</a></li>
    </ul>
</nav>
```

→ Possibilité de créer un composant menu

ANGULAR : ROUTING

Comment naviguer dans l'application ?

1. A travers des liens : **RouterLink**

→ Cas pour une **URL dynamique**: → **/first-composant/:id**

/second-composant/:name

```
<nav>
    <ul>
      <li><a [routerLink]="['/first-composant', monObjet.id]" routerLinkActive="active">First Component ID</a></li>
      <li><a [routerLink]="['/second-composant', monObjet.name]" routerLinkActive="active">Second Component Name</a></li>
    </ul>
</nav>
```

ANGULAR : ROUTING

Comment naviguer dans l'application ?

1. A travers des liens : **RouterLink**

→ Cas pour une **URL dynamique**: → **/first-composant?id=1&nom:Zelda**

```
<a [routerLink]="[/personnage]" [queryParams]="{ id: '1', nom: 'Zelda'}">Personnage Zelda</a>
```

ANGULAR : ROUTING

Comment naviguer dans l'application ?

2. **Directement dans le code du composant**

→ Besoin d'injecter une instance de `Router` dans le composant

```
constructor(private router: Router) {}
```

→ Utilisation de la méthode **navigate / navigateByUrl**

```
goBack() {
  this.router.navigate(['previousComposant']);
}
```

Utile pour les retours !

ANGULAR : ROUTING

Comment naviguer dans l'application ?

2. Directement dans le code du composant

→ Autre possibilité pour aller à la vue précédente :
Utilisation du service **Location** de Angular.

```
constructor(private location: Location) {  }
```

```
goBack(): void {
  this.location.back();
}
```

ANGULAR : ROUTING

Récupération des données de routage ?

Possibilité d'utiliser une route pour passer des informations d'un composant à un autre.

→ Utilisation de l'interface `ActivatedRoute`.

→ Utilisation d'un objet de cette classe dans la méthode **`ngOnInit()`**

1) Les propriétés **snapshot.params / paramMap** contiennent **tous les paramètres passés à la route**.

Solution avec les snapshot (instantanée)

Solution avec les observables (asynchrone)

ANGULAR : ROUTING

Comment récupérer « 1 » dans l'URL /personnage/1 ?

Récupération des données de routage ?

→ Récupération de la route de la forme ***personnage/:id***


```
constructor(private route: ActivatedRoute,
  private router: Router, private persoService:PersonnageService) {}

ngOnInit(): void {
    const id = +this.route.snapshot.params.id;  //+ pour caster id en number
    this.persoService.getPersonnageObservable(id).subscribe(perso => this.perso = perso);
    //Observable pour récupérer le personnage avec le bon id
}
```

Snapshot.params

→ Paramètres de routage passés sous forme de chaînes de caractères.

ANGULAR : ROUTING

Comment récupérer « 1 » dans l'URL /personnage/1 ?

Récupération des données de routage ?

→ Récupération de la route de la forme ***personnage/:id***


```
constructor(private route: ActivatedRoute,
  private router: Router, private persoService:PersonnageService) {}

ngOnInit(): void {
  this.route.paramMap.subscribe( res => {
    const id = +res.get('id')!;
    this.perso = this.persoService.getPersonnage(id);
  });
}
```

paramMap

→ Paramètres de routage passés sous forme de chaînes de caractères.

ANGULAR : ROUTING

Récupération des données de routage ?

Possibilité d'utiliser une route pour passer des informations d'un composant à un autre.

→ Utilisation de l'interface `ActivatedRoute`.

→ Utilisation d'un objet de cette classe dans la méthode **`ngOnInit()`**

2) Les propriétés **snapshot.queryParams / queryParamMap** permettent de récupérer les paramètres de l'URL **sans définition d'une route**.

Solution avec les snapshot (instantanée)

Solution avec les observables (asynchrone)

ANGULAR : ROUTING

Récupération des données de routage ?

→ Récupération de la route de la forme ***personnage?id=value1&name=value2***


```
constructor(private route: ActivatedRoute,
  private router: Router, private persoService:PersonnageService) {}

ngOnInit(): void {
  this.idQuery = this.route.snapshot.queryParams.id;
  this.nomQuery = this.route.snapshot.queryParams.nom;
  console.log(`queryParam : id = ${this.idQuery} et nom = ${this.nomQuery}`);
}
```

Snapshot.queryParams

ANGULAR : ROUTING

Récupération des données de routage ?

→ Récupération de la route de la forme ***personnage?id=value1&name=value2***


```
constructor(private route: ActivatedRoute,
  private router: Router, private persoService:PersonnageService) {}

ngOnInit(): void {
    this.route.queryParamMap.subscribe(res => {
      this.idQuery = res.get('id') ?? '';
      this.nomQuery = res.get('nom') ?? '';
    });
}
```

queryParamMap

ANGULAR : ROUTING

Récupération des données de routage ?

Quelle méthode choisir ?

→ **Snapshot** : Si la valeur initiale des paramètres est utilisée seulement à l'initialisation du composant et ne risque pas de changer.

→ **Observable** : Si la route risque de changer tout en restant dans le même composant (l'initialisation du composant à travers *ngOnInit()* ne sera pas appelée à nouveau mais l'observateur sera notifié lorsque l'URL est modifiée).

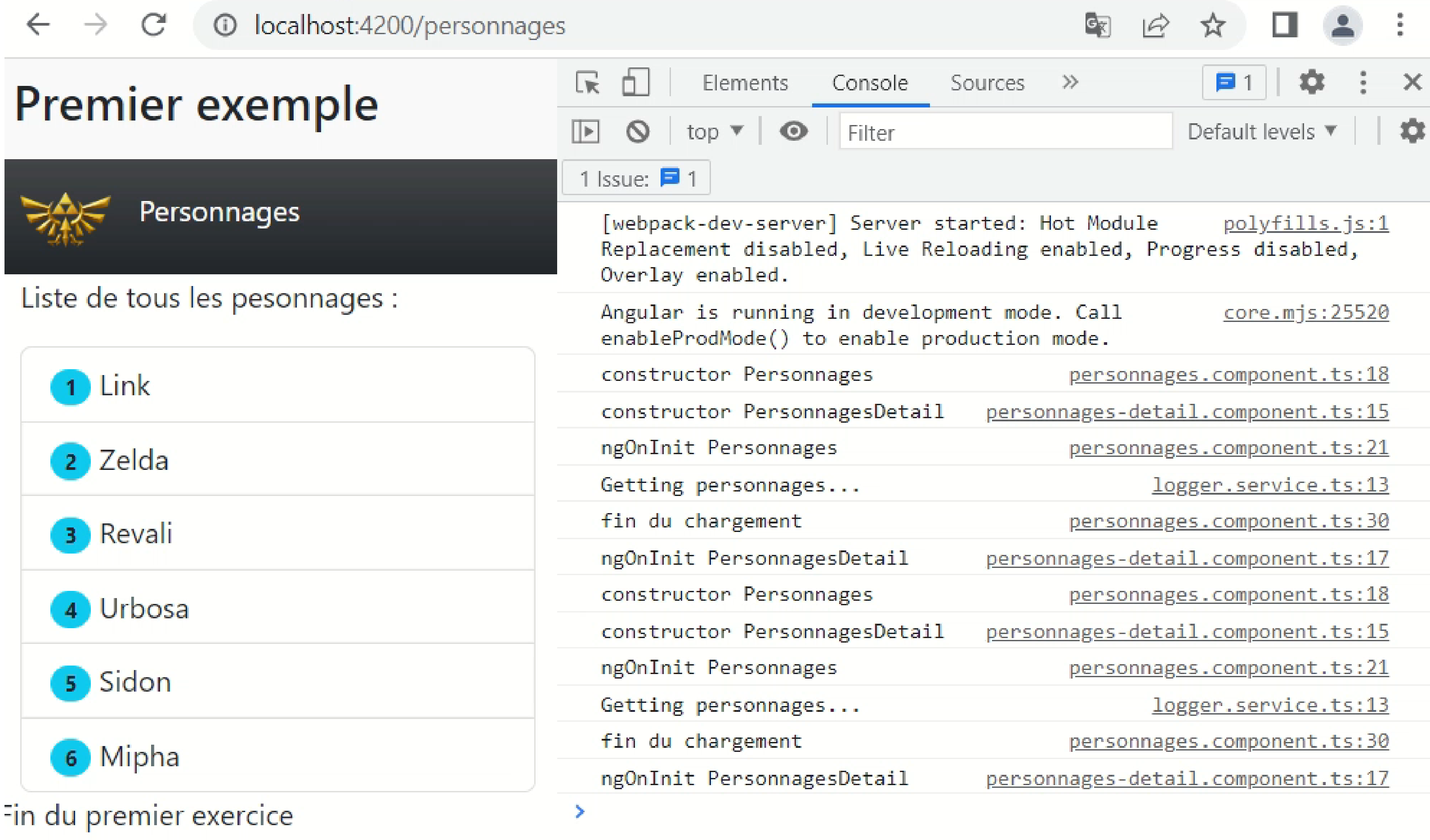
ANGULAR : EXEMPLE

Exemple de résultat de navigation :

→ Création d’une barre de menu avec `routerLink` (/personnages)

```
Cours > jxc-cours > src > app > nav > <> nav.component.html > ...
<nav class="navbar navbar-expand-sm bg-dark navbar-dark bg-gradient bg-opacity-85 navbar-index sticky-top">
    <div class="container-fluid">
        <a class="navbar-brand" [routerLink]="['/home']" routerLinkActive="active">
            <img src="../assets/img/triforce.png" alt="Home" width="50">
        </a>
        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
            <li class="nav-item">
                <a class="nav-link" aria-current="page"
                [routerLink]="['/personnages']" routerLinkActive="active">Personnages</a>
            </li>
        </ul>
    </div>
</nav>
```

ANGULAR : EXEMPLE



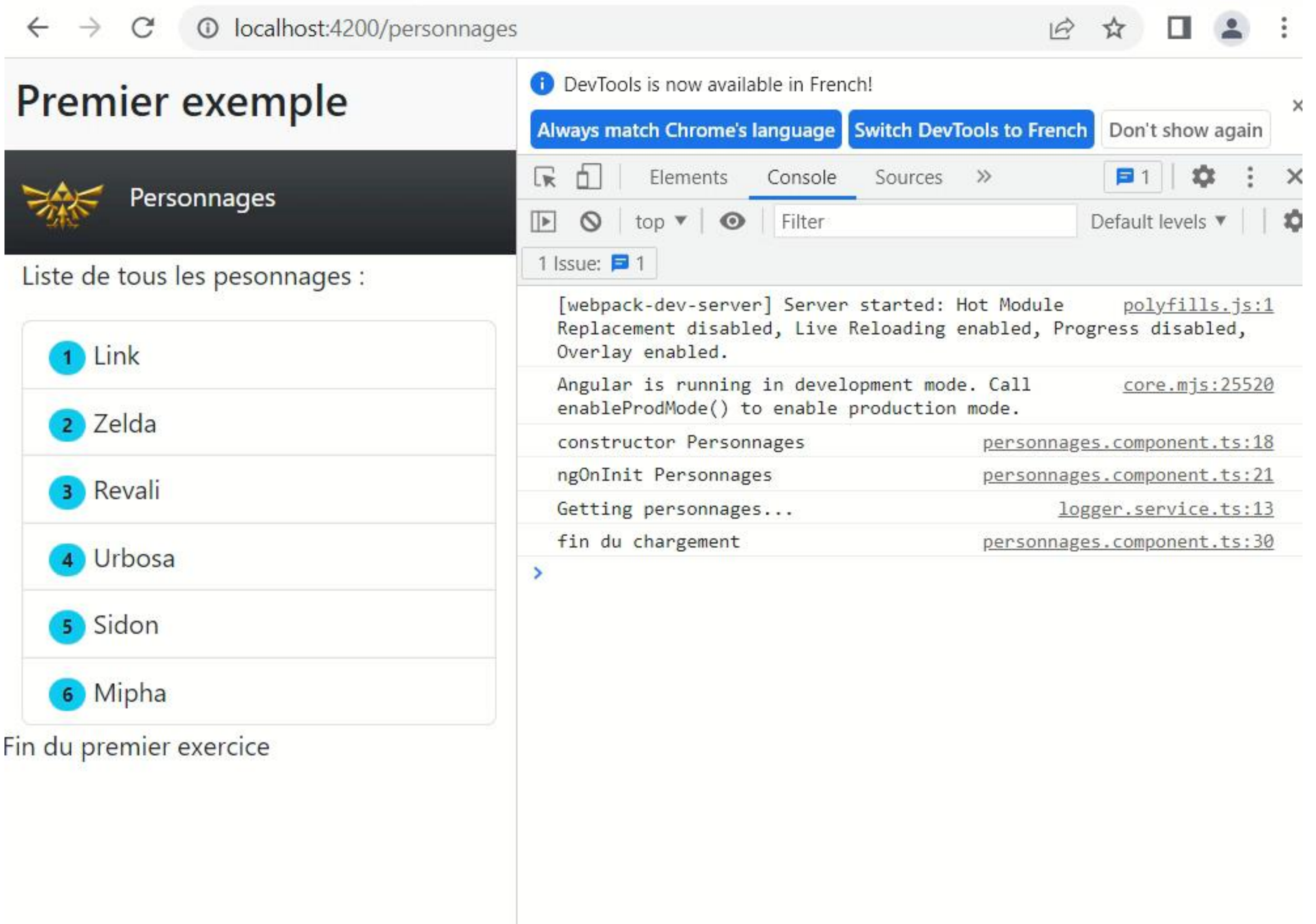
ANGULAR : EXEMPLE

Exemple de résultat de navigation :

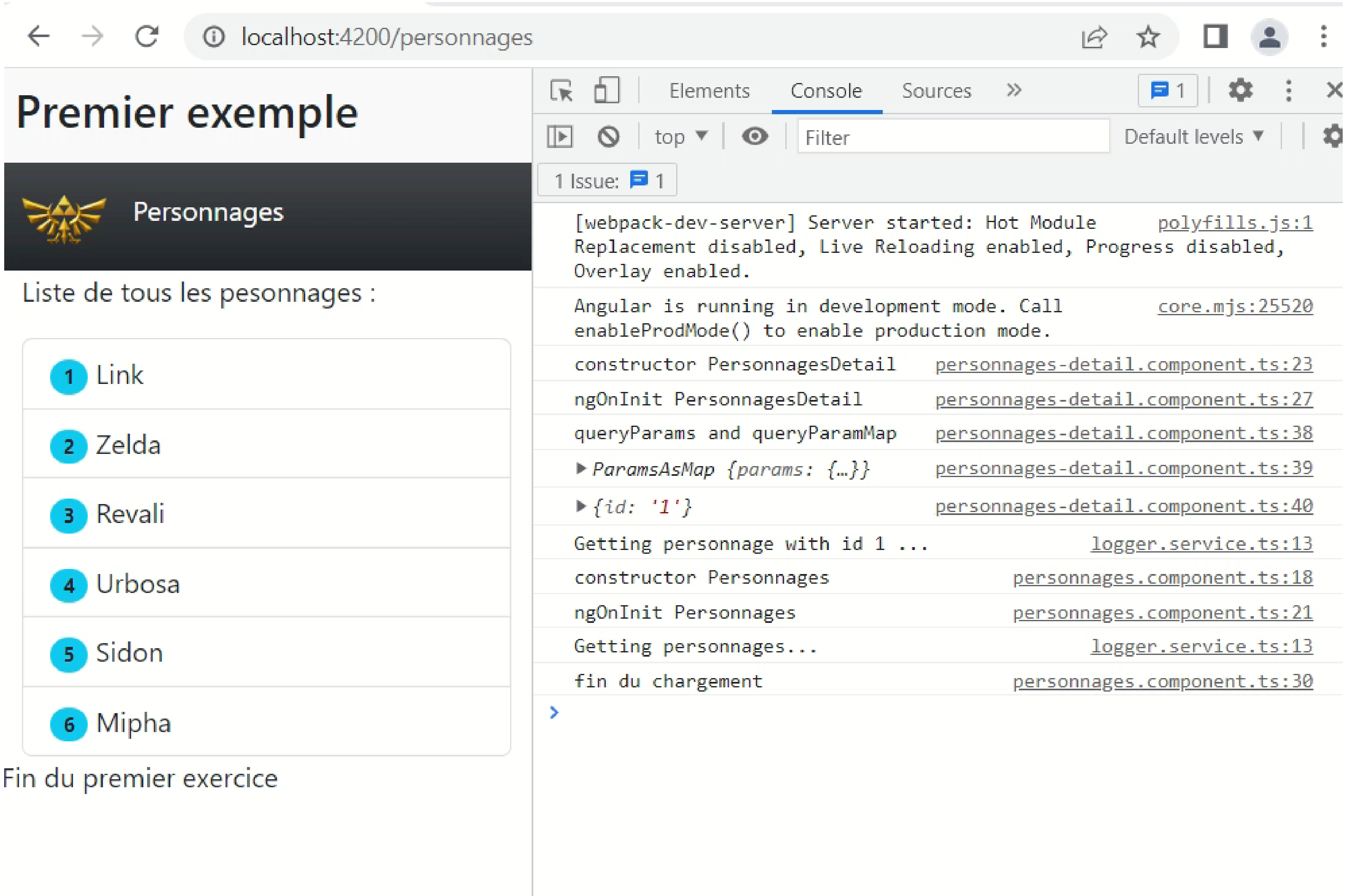
→ Lien dans le composant Personnage avec un `routerLink` et un id

```
urs > jxc-cours > src > app > personnages > <> personnages.component.html > ...
<div class="container">
    <p>Liste de tous les pesonnages :</p>
    <ul class="list-group personnages">
        <li *ngFor="let hero of listPersonnages"
            class="list-group-item"
            [class.selected]="hero===myHero"
            (click)="selectedPersonnage(hero)"
            [routerLink]="['/hero', hero.id]" routerLinkActive="active">
            <span class="badge rounded-pill bg-info text-dark">{{hero.id}}</span>
            {{hero.name}}
            <span *ngFor="let arme of hero.arme"> {{arme}}</span>
    </ul>
</div>
<!-- <app-personnages-detail [myHero]="myHero" (newArmeEvent)="modifierArme($event)"></app-personnages-detail>
```

ANGULAR : EXEMPLE



ANGULAR : EXEMPLE



ANGULAR : ROUTING

Resolve

→ Lors de l'utilisation d'API, il peut y avoir un délai avant que les données qui doivent être affichées soient retournées du serveur (affichage d'une page blanche ou d'un message par défaut).

Utiliser **Resolve** permet :

→ d'attendre le retour d'un observable avant d'initialiser ou mettre à jour un composant après une mise à jour de l'url.

→ de passer un paramètre dynamique à une vue dans une route.

ANGULAR : ROUTING

ours > jxc-cours > src > app > personnages-detail > TS personnage.resolver.ts > ...

```
import { Injectable } from '@angular/core';
import { Resolve, RouterStateSnapshot, ActivatedRouteSnapshot } from '@angular/router';
import { Observable } from 'rxjs';
import { PersonnageService } from '../services/personnage.service';
import { IPersonnage } from '../personnages/IPersonnage';

@Injectable({
  providedIn: 'root'
})
export class PersonnageResolver implements Resolve<IPersonnage> {

  constructor(private personnageService: PersonnageService) { }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<IPersonnage> {
    console.log("resolve ...");
    const id = route.params['name'];
    return this.personnageService.getPersonnageById(+id);
  }
}
```

ANGULAR : ROUTING

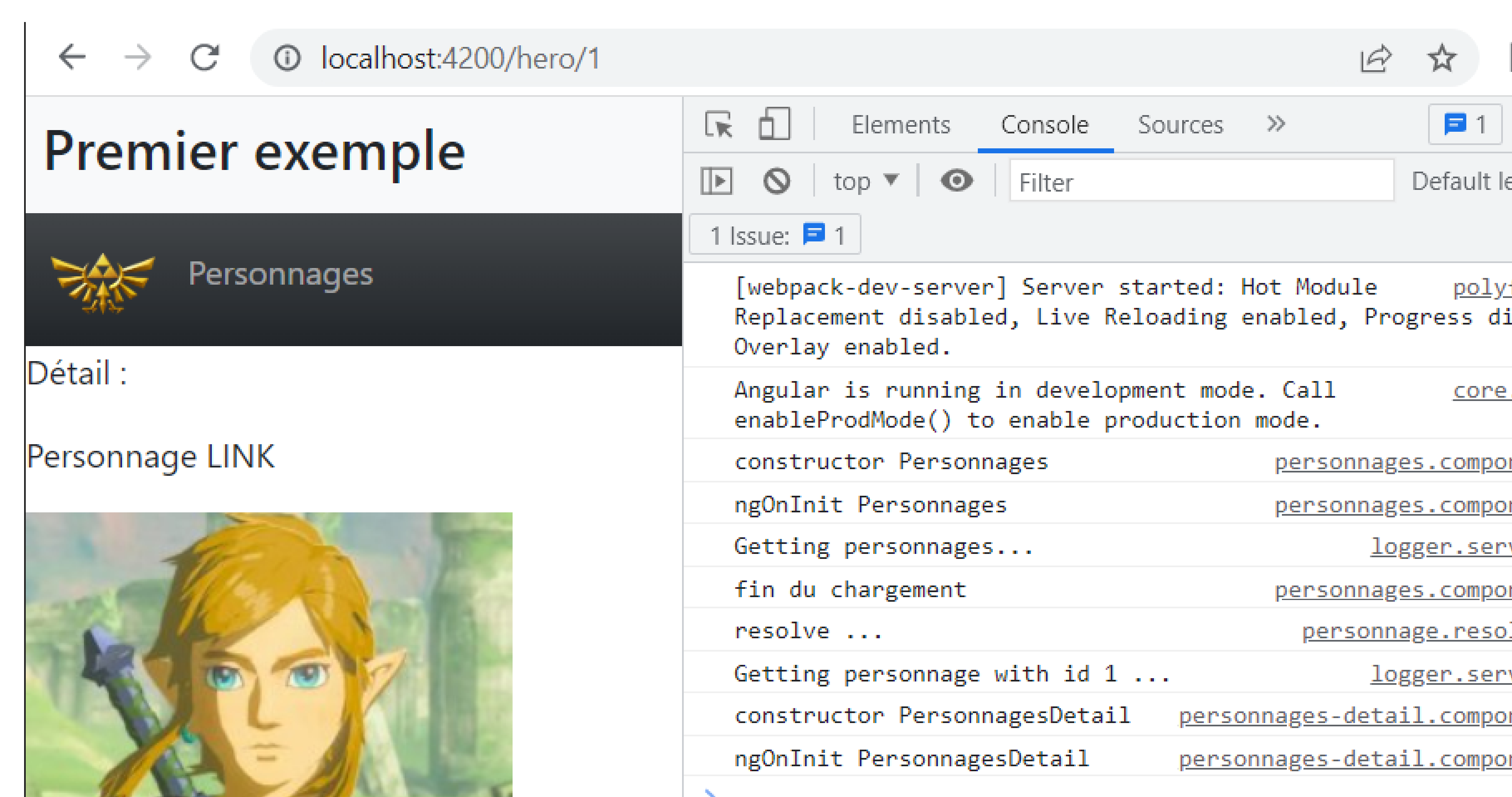
Resolve

Intégration dans la route :

```
const routes: Routes = [
  {path: 'hero/:name',
    component: PersonnagesDetailComponent,
    resolve: {hero: PersonnageResolver}
  },
```


Récupération des données :

```
ngOnInit(): void {
  console.log("ngOnInit PersonnagesDetail");
  this.route.data.subscribe(
    data => { this.myHero = data['hero']}
  );
  if(!this.myHero) this.router.navigate(['/personnages']);
}
```


localhost:4200/hero/1
Premier exemple
Personnages
Détail :
Personnage LINK
Quelle arme ?
Modifier
Fin du premier exercice
Elements
Console
Sources
top
Filter
Default levels
1 Issue: 1
[webpack-dev-server] Server started: Hot Module Replacement disabled, Live Reloading enabled, Progress disabled, Overlay enabled.
polyfills.js:1
Angular is running in development mode. Call enableProdMode() to enable production mode.
core.mjs:25520
constructor Personnages
personnages.component.ts:18
ngOnInit Personnages
personnages.component.ts:21
Getting personnages...
logger.service.ts:13
fin du chargement
personnages.component.ts:30
resolve ...
personnage.resolver.ts:19
Getting personnage with id 1 ...
logger.service.ts:13
constructor PersonnagesDetail
personnages-detail.component.ts:23
ngOnInit PersonnagesDetail
personnages-detail.component.ts:27

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Observable
- Rooting
- **Forms**

ANGULAR : FORMULAIRE

→ Outils graphiques créé avec le langage **HTML**.

→ Permet à l'utilisateur d'interagir avec les données de l'application (se connecter, entrer des informations dans la base de données, mettre à jour un profil, etc.).

→ **Angular facilite la gestion d'un formulaire** :

- La récupération des données saisies,
- La validation et le contrôle des valeurs saisies,
- La gestion d'erreur,
- Et bien d'autres.

ANGULAR : FORMULAIRE

Angular propose deux approches :

→ **Template-driven forms**

- Basé sur `FormsModule`.
- Facile à utiliser et conseillé pour les formulaires simples.

→ **Reactive forms**

- Basé sur `ReactiveFormsModule`.
- Robuste et évolutif, conçu pour des applications nécessitant des contrôles particuliers (`Form Group` et `Form Builder`).

ANGULAR : FORMULAIRE

Différences principales entres les deux approches :

	REACTIVE	TEMPLATE-DRIVEN
Setup of form model	Explicit, created in component class	Implicit, created by directives
Data model	Structured and immutable	Unstructured and mutable
Data flow	Synchronous	Asynchronous
Form validation	Functions	Directives

https://angular.io

ANGULAR : FORMULAIRE

Les **exemples suivants** seront basés sur une saisie de produits.

→ Interface **IProduit** :

```
exempleForm > src > app > TS IProduit.ts > ...
1  export interface IProduit{
2      id: number,
3      name: string,
4      category: string
5  }
```


→ Composant **Products** :

```
exempleForm > src > app > products > TS products.component.ts > ...
1   import { Component, OnInit } from '@angular/core';
2   import { IProduit } from '../IProduit';
3
4   @Component({
5     selector: 'app-products',
6     templateUrl: './products.component.html',
7     styleUrls: ['./products.component.css']
8   })
9   export class ProductsComponent implements OnInit {
10
11    produits: IProduit[] = [];
```

```
exempleForm > src > app > products > <> products.component.html > ...
1  <h2>Liste des produit :</h2>
2  <ul>
3      <li *ngFor="let produit of produits">
4          {{ produit.name }} {{ produit.category }}
5      </li>
6  </ul>
```

ANGULAR : TEMPLATE-DRIVEN FORMS

Première approche : Template-driven forms

→ Utilise la directive `ngModel` pour réaliser une **liaison bidirectionnelle**.

→ Cette directive permet de créer et manager une instance de `FromControl` pour un élément donné du formulaire.

→ Besoin d'importer `FormsModule` dans `app.module.ts` afin d'utiliser `ngModel` :

```
import { FormsModule } from '@angular/forms';
```

ANGULAR : TEMPLATE-DRIVEN FORMS

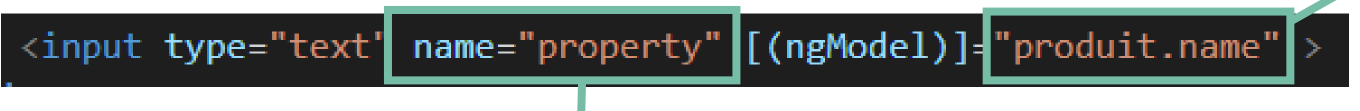
Création d'un composant :

exempleForm > src > app > form-td > TS form-td.component.ts > ...

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

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

  categories = ["Légumes", "Fruits", "Viandes"];
  produit!: IProduit;

  constructor() { }

  ngOnInit(): void {
    this.produit = {
      id: 0,
      name: 'Pas de nom',
      category: this.categories[0]
    };
  }
}
```

ANGULAR : TEMPLATE-DRIVEN FORMS

Syntaxe ngModel :



Dans le composant il existe une propriété `name` à l'objet `produit`.

→ Angular crée des `FormControls` et les enregistres avec une directive `ngForm` qu'Angular attache à une balise `form`.
Chaque `FormControl` est enregistré avec le nom de l'input associé (`name`).

→ Modification directe de la propriété `name` avec la nouvelle valeur saisie.

ANGULAR : TEMPLATE-DRIVEN FORMS

Exemple :

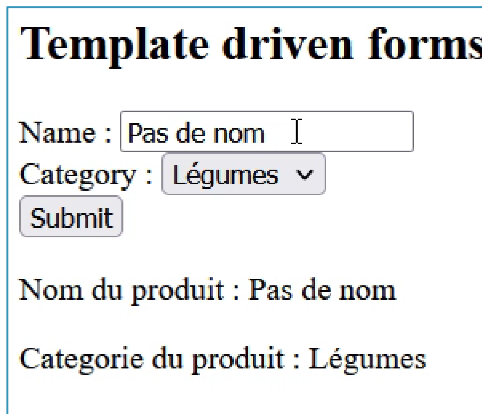
Un premier formulaire avec :

→ un champ texte pour saisir le nom du produit

→ un champ select pour sélectionner la catégorie du produit

→ un bouton submit

Résultat :



ANGULAR : TEMPLATE-DRIVEN FORMS

Directive `ngForm` attatchée à la balise `form`

(utilisation *template reference variable*)

exempleForm > src > app > form-td > form-td.component.html > ...

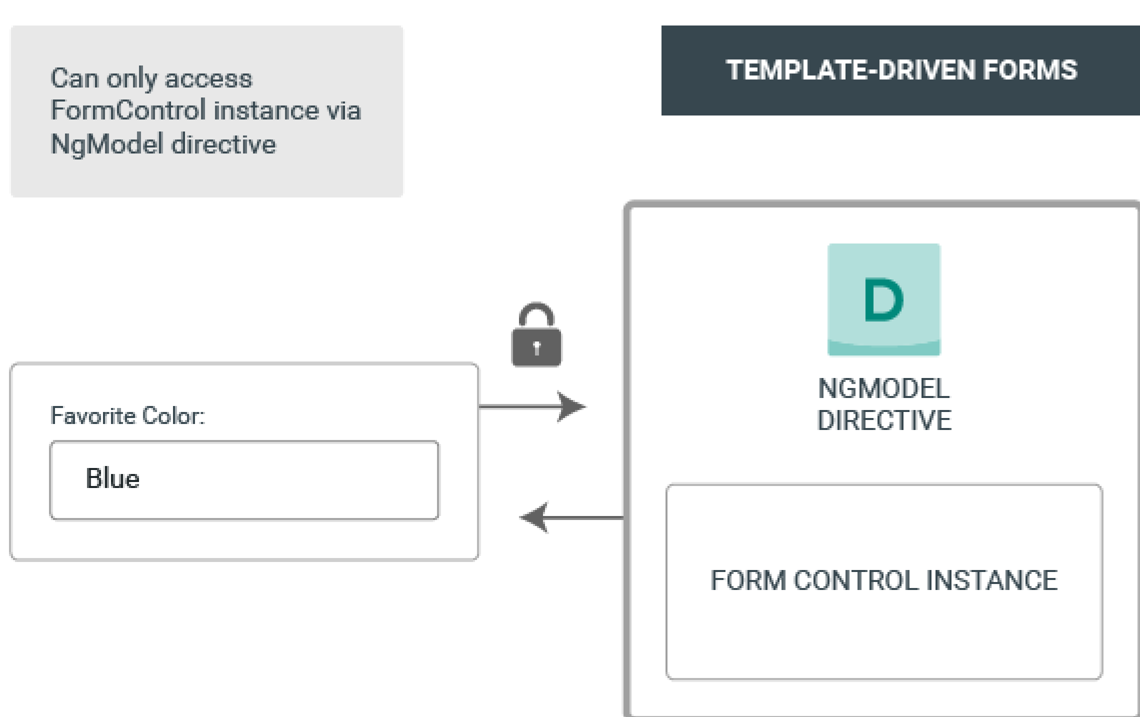
```
<h2>Template driven forms</h2>
<form #productFormTD="ngForm">
    <div class="group">
        <label for="name">Name : </label>
        <input type="text" name="name" id="name" [(ngModel)]="produit.name" >
    </div>
    <div class="group">
        <label for="category">Category : </label>
        <select id="category" name="category" [(ngModel)]="produit.category" >
            <option *ngFor="let cat of categories" [value]="cat">{{cat}}</option>
        </select>
    </div>
    <button type="submit">Submit</button>
    <div>
        <p>Nom du produit : {{produit.name}}</p>
        <p>Categorie du produit : {{produit.category}}</p>
    </div>
</form>
```

La propriété `produit.category` est lié au `select` avec `ngModel`.

Pas besoin de cliquer pour envoyer la valeur saisie dans le champ texte.

ANGULAR : TEMPLATE-DRIVEN FORMS

→ Avec ce type d'approche nous n'avons **pas un accès direct** à l'instance du **`FromControl`**.

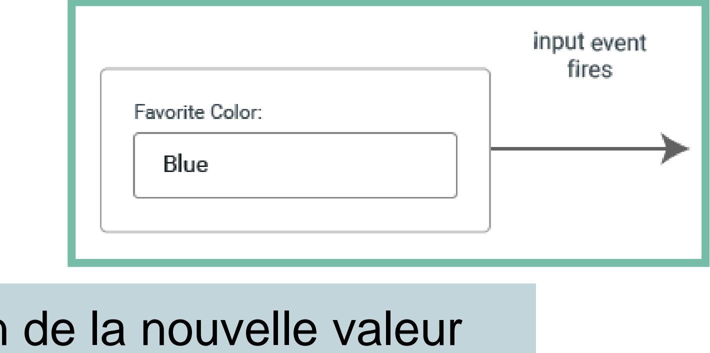


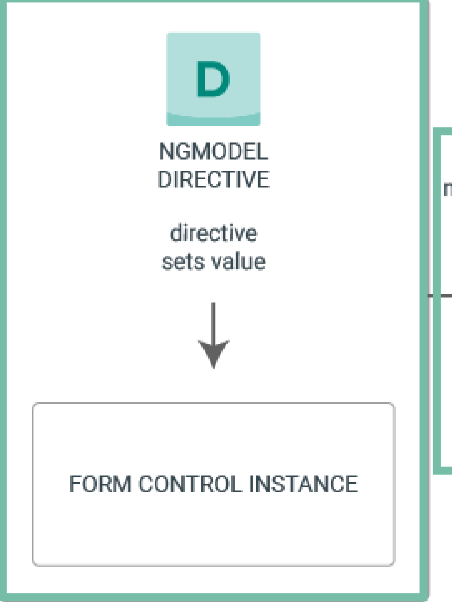
https://angular.io

ANGULAR : TEMPLATE-DRIVEN FORMS

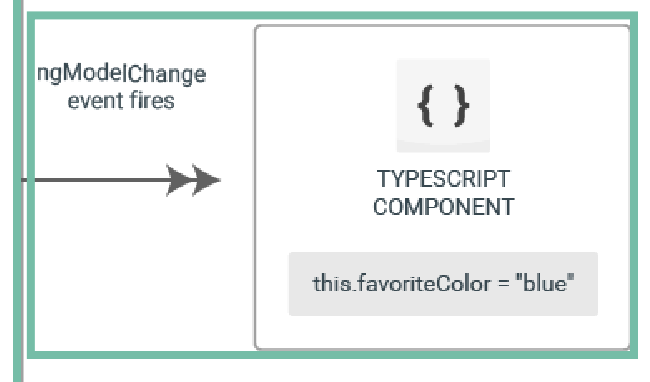
TEMPLATE-DRIVEN FORMS - DATA FLOW (VIEW TO MODEL)

1. L'élément Input émet un **`InputEvent`**





3. `ngModel.viewToModelUpdate()` est appelée et émet un `ngModelChange Event.`



Emission de la nouvelle valeur via l'observable valueChanges

2. Déclanchement de la méthode `setValue()` sur l'instance de `FormControl`

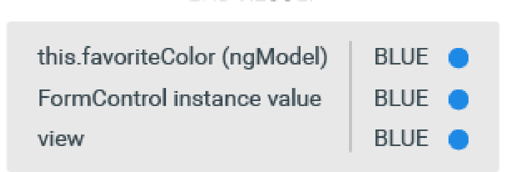
START

this.favoriteColor (ngModel)	RED
FormControl instance value	RED
view	BLUE

DIRECTIVE SETS VALUE

this.favoriteColor (ngModel)	RED
FormControl instance value	BLUE
view	BLUE

END RESULT



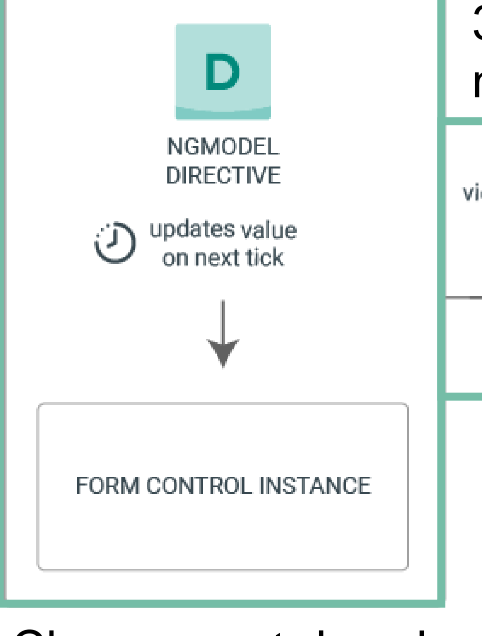
this.favoriteColor (ngModel)	BLUE
FormControl instance value	BLUE
view	BLUE

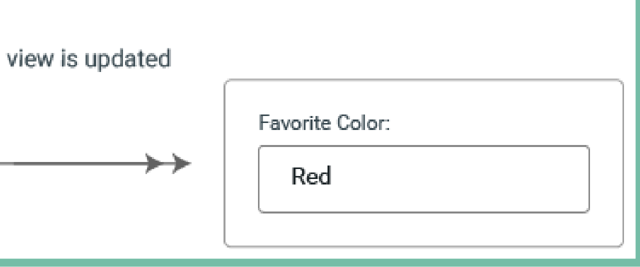
https://angular.io

ANGULAR : TEMPLATE-DRIVEN FORMS

1. La méthode `ngOnChanges` est appelée suite à une modification d'une valeur (directive `ngModel`).

TEMPLATE-DRIVEN FORMS - DATA FLOW (MODEL TO VIEW)





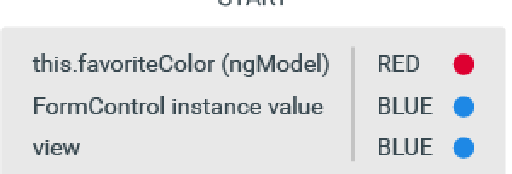
3. L'instance de `FormControl` émet la valeur modifiée via l'observable `valueChanges`.

Modification de l'Input avec la nouvelle valeur.

Mise dans une file d'attente la demande asynchrone de modification de valeur du FormControl.

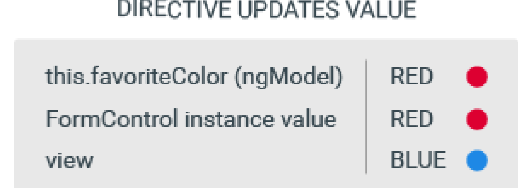
2. Changement de valeur

START



this.favoriteColor (ngModel)	RED	🔴
FormControl instance value	BLUE	🔵
view	BLUE	🔵

DIRECTIVE UPDATES VALUE



this.favoriteColor (ngModel)	RED	🔴
FormControl instance value	RED	🔴
view	BLUE	🔵

END RESULT

this.favoriteColor (ngModel)	RED	🔴
FormControl instance value	RED	🔴
view	RED	🔴

ANGULAR : TEMPLATE-DRIVEN FORMS

→ L'utilisation de la directive `ngModel` permet donc de suivre l'état d'un input à un moment donné.

→ Angular propose différentes classes CSS pour gérer les différents états :

State	Class if true	Class if false
The control has been visited.	`ng-touched`	`ng-untouched`
The control's value has changed.	`ng-dirty`	`ng-pristine`
The control's value is valid.	`ng-valid`	`ng-invalid`

https://angular.io

ANGULAR : TEMPLATE-DRIVEN FORMS

Utilisation du couple `ng-valid` / `ng-invalid` :

→ CSS :

exempleForm > src > app > form-td > # form-td.component.css >

```css
.ng-valid[required] {
    border-left: 5px solid #357939;
}

.ng-invalid:not(form)  {
    border-left: 5px solid #88302e;
}
```

Name :
Category : Légumes

Name : Tomate
Category : Légumes

→ Modification du formulaire :

exempleForm > src > app > form-td > <> form-td.component.html > ...

```html
<h2>Template driven forms</h2>
<form #productFormTD="ngForm">
    <div class="group">
        <label for="name">Name : </label>
        <input type="text" name="name" id="name" [(ngModel)]="produit.name" required>
    </div>
```

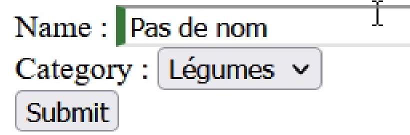
ANGULAR : TEMPLATE-DRIVEN FORMS

Soumission du formulaire

→ La directive `ngForm` possède une notion de validité du formulaire.

→ Il est par exemple possible de désactiver le bouton *submit* tant que le formulaire n'est pas valide :

```
<button type="submit" [disabled]=!productFormTD.valid>Submit</button>
```



→ Possibilité de créer ses propres validateurs.

ANGULAR : TEMPLATE-DRIVEN FORMS

Soumission du formulaire

→ La directive `ngSubmit` permet de soumettre le formulaire: (event binding)


```html
<form #productFormTD="ngForm" (ngSubmit)=ajouterProduit()>
```

exempleForm > src > app > form-td > TS form-td.component.ts > ...

```typescript
27  ajouterProduit(): void{
28    this.produits.push(this.produit);
29    this.produit = { id: 0, name: 'Pas de nom', category: this.categories[0] }
30  }
```

ANGULAR : REACTIVE FORM

Deuxième approche : Reactive form

→ A l'inverse de la première approche, nous allons pouvoir manipuler les contrôles de chaque entité du formulaire.

→ Définition des contrôles directement dans le composant :
La directive `[formControl]` va directement lier l'instance de `FormControl` à un élément de la vue.

→ Besoin d'importer le module : **`ReactiveFormsModule`**

```
import { ReactiveFormsModule } from '@angular/forms';
```

ANGULAR : REACTIVE FORM

Instance **`FormControl`** :

exempleForm > src > app > form-r > TS form-r.component.ts > ...

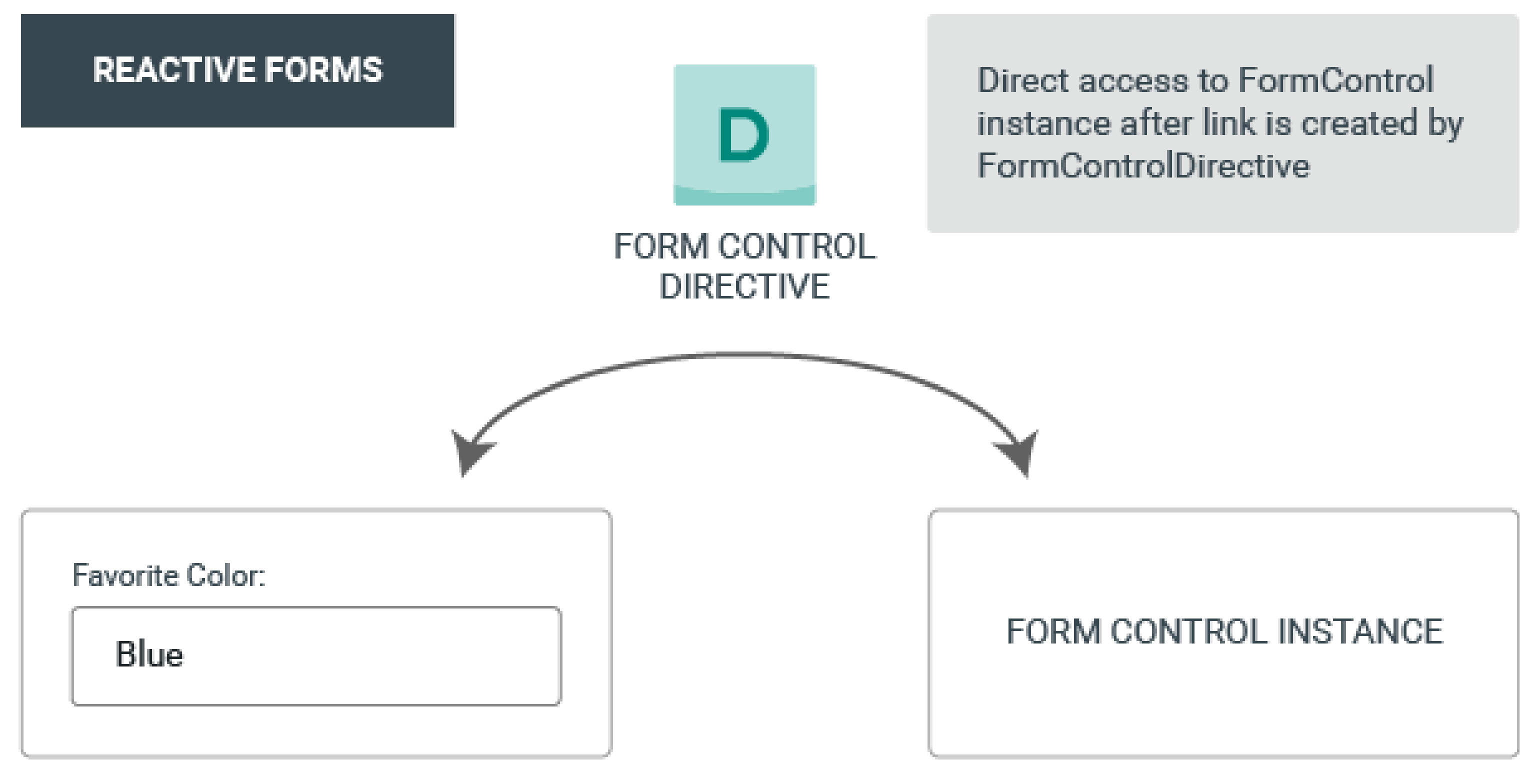
```
import { Component, OnInit } from '@angular/core';
import { FormControl } from '@angular/forms';

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

  name: string = 'defaut'
  nameControl = new FormControl('defaut');

  constructor() { }

  ngOnInit(): void {
  }
}
```

ANGULAR : REACTIVE FORM

Syntaxe de la directive **`formControl`** :

```
exempleForm > src > app > form-r > <> form-r.component.html > ...
1   <h2>Template reactive form</h2>
2   <form #productFormTD="ngForm">
3       <div class="group">
4           <label for="name">Name : </label>
5           <input type="text" name="name" id="name" [formControl]="nameControl">
6       </div>
7       <div>
8           <p>Nom du produit (control) : {{nameControl.value}}</p>
9           <p>Nom du produit (data) : {{name}}</p>
10      </div>
11  </form>
```

ANGULAR : REACTIVE FORM



https://angular.io

ANGULAR : REACTIVE FORM

→ L'instance de `FormControl` à accès à la valeur courante de l'input associé.

→ **Aucune modification** de la donnée du composant est réalisé directement.

→ Les maj de la vue au modèle et inversement du modèle à la vue sont synchrones.

Template reactive form

Name : defaut

Nom du produit (control) : defaut

Nom du produit (data) : defaut

ANGULAR : REACTIVE FORM

REACTIVE FORMS - DATA FLOW (VIEW TO MODEL)

D

FORM CONTROL DIRECTIVE

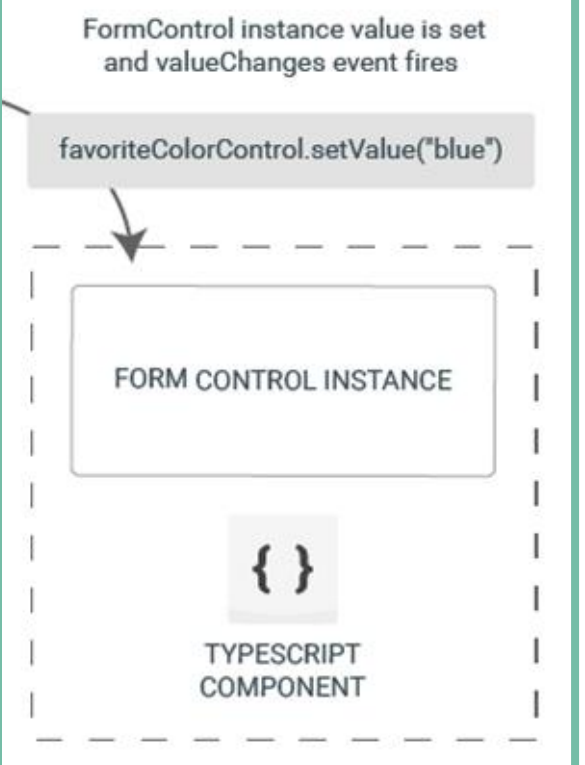
1. L’utilisateur saisit la nouvelle valeur. L’élément Input émet un `InputEvent`.

2. La nouvelle valeur est immédiatement transmise à l’instance de `FormControl`.

input event fires

Favorite Color:

Blue



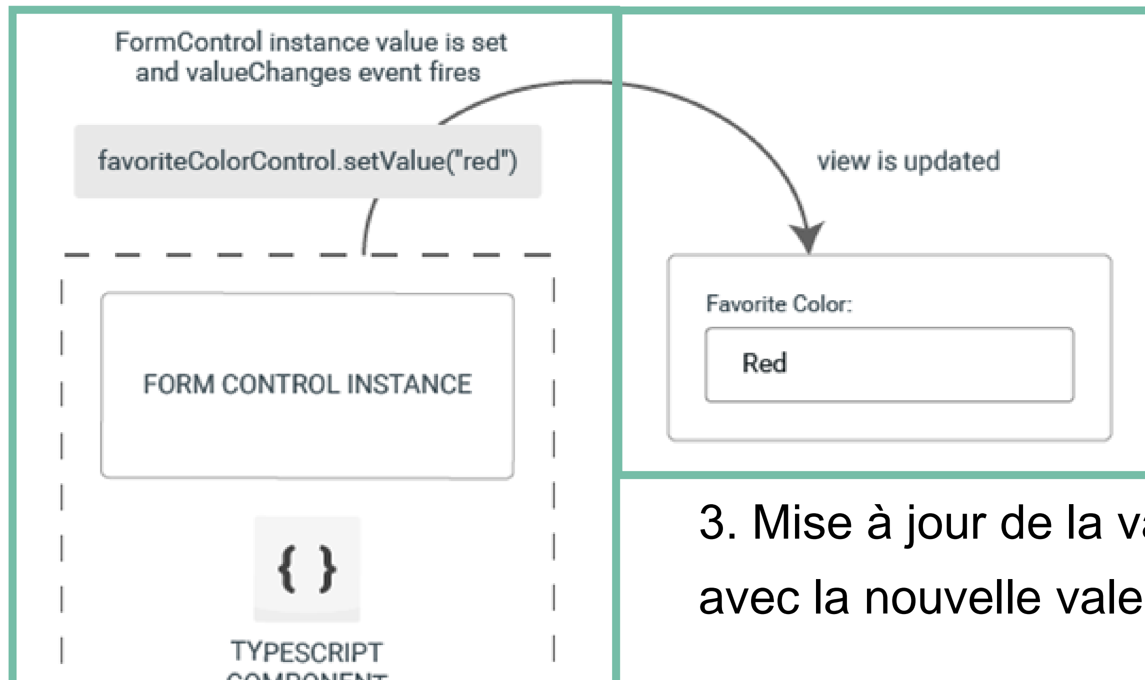
3. L’instance de `FormControl` ément la nouvelle valeur via l’observable valueChanges.

https://angular.io

ANGULAR : REACTIVE FORM

REACTIVE FORMS - DATA FLOW (MODEL TO VIEW)

1. L'utilisateur appelle la méthode `setValue` qui met à jour la valeur du `FormControl`.



2. L'instance de `FormControl` émet la nouvelle valeur avec l'observable `valueChanges`.

→ Une souscription à cet observable permet de recevoir la nouvelle valeur.

3. Mise à jour de la valeur de l'Input avec la nouvelle valeur.

https://angular.io

ANGULAR : REACTIVE FORM

Souscription à l'observable `valueChanges` pour récupérer la nouvelle valeur :

exempleForm > src > app > form-r > TS form-r.component.ts > ...

```
 9  export class FormRComponent implements OnInit {
10
11    name: string = 'defaut'
12    nameControl = new FormControl('defaut');
13
14    constructor() { }
15
16    ngOnInit(): void {
17      this.nameControl.valueChanges.subscribe( res =>
18        this.name = res
19      )
20    }
21  }
```


Template reactive form

Name : defaut

Nom du produit (control) : defaut

Nom du produit (data) : defaut

ANGULAR : REACTIVE FORM

Possibilité de regrouper plusieurs **`FormControl`** dans un **`FormGroup`**.

```
exempleForm > src > app > form-r > TS form-r.component.ts > FormRComponent >
 9  export class FormRComponent implements OnInit {
10
11    name: string = 'defaut';
12    category: string = '';
13    categories = ["", "Légumes", "Fruits", "Viandes"];
14
15    //nameControl = new FormControl('defaut');
16    produitForm = new FormGroup({
17      nameControl: new FormControl(''),
18      categoryControl: new FormControl(this.categories[0]),
19    });
```

→ Peut également imbriquer plusieurs `FromGroup`.

ANGULAR : REACTIVE FORM

formGroupName=…
Si imbrication d’un autre fromGroup

exempleForm > src > app > form-r > form-r.component.html > ...

```html
<h2>Template reactive form</h2>
<form #productFormTD="ngForm" [formGroup]="produitForm">
    <div class="group">
        <label for="name">Name : </label>
        <input type="text" name="name" id="name" formControlName="nameControl">
    </div>
    <div class="group">
        <label for="category">Category : </label>
        <select id="category" name="category" formControlName="categoryControl" >
            <option *ngFor="let cat of categories" [value]="cat">{{cat}}</option>
        </select>
    </div>
    <div>
        <p>Nom du produit : {{name}}</p>
        <p>Categorie : {{category}}</p>
    </div>
</form>
```

ANGULAR : REACTIVE FORM

Template reactive form

Name :
Category :

Nom du produit : defaut

Categorie :

Possibilité également de souscrire à l'observable `valueChange` pour chacun des `FormControl` du groupe :

```
ngOnInit(): void {
  this.produitForm.get('nameControl')?.valueChanges.subscribe( res =>
    this.name = res
  )
  this.produitForm.get('categoryControl')?.valueChanges.subscribe( res =>
    this.category = res
  )
}
```

→ Le groupe traque chaque modification de ses contrôles. Lorsqu'un contrôle change, le parent émet également une nouvelle valeur.

ANGULAR : REACTIVE FORM

Comment mettre à jour une valeur du modèle (formGroup) ?

→ Utilisation de `setValue()` ou de `patchValue()`


```
updateProduit(): void{
  this.produitForm.patchValue({
    nameControl: "Toto"
  })
  this.produitForm.setValue({
    nameControl: "Toto",
    categoryControl: this.categories[1]
  })
}
```

ANGULAR : REACTIVE FORM

Soumission du formulaire

Soumission du formulaire si celui-ci est valide (par rapport au groupe)

```
<button type="submit" [disabled]="!produitForm.valid">Submit</button>
```


Comme pour les te*mplate-driven forms* : utilisation de la directive `ngSubmit`.

```
<form [formGroup]="produitForm" (ngSubmit)="ajouterProduit()">
```

```
ajouterProduit(): void{
  console.dir(this.produitForm.value);
  this.produit.name = this.name;
  this.produit.category = this.category;
  this.produits.push(this.produit);
}
```

ANGULAR : REACTIVE FORM

→ Possibilité d'utiliser le service **FormBuilder** pour faciliter la création des contrôles.

```
import { FormBuilder } from '@angular/forms';
```



```
exempleForm > src > app > form-r > TS form-r.component.ts > ...
11    export class FormRComponent implements OnInit {
12
13      produitFormFB = this.fb.group({
14        nameControl: [''],
15        categoryControl: ['']
16      })
17
18      constructor(private fb: FormBuilder) { }
```

ANGULAR : REACTIVE FORM

Fonctions de validation

→ Possibilité d'ajouter directement à un `formControl` des fonctions de validations.

- **Validateurs synchrones** (retourne directement les erreurs ou null, 2ème paramètre)
- **Validateurs asynchrones** (retourne un observable qui émet plus tard les erreurs ou null, 3ème paramètre)

→ Les validateurs asynchrones sont réalisés que si les validateurs synchrones retournent aucun problème.

ANGULAR : REACTIVE FORM

Fonctions de validation

→ Angular propose des fonctions de validation : voir la classe **`Validators`**.

required, maxLenght, minLenght, pattern, etc.


```
import { Validators } from '@angular/forms';
```

```
produitForm = new FormGroup({
  nameControl: new FormControl('', [Validators.required, Validators.minLength(3)]),
  categoryControl: new FormControl(this.categories[0])
});
```

ANGULAR : REACTIVE FORM

Fonctions de validation

Affichage du message d'erreur dans le template :

```
<div class="group">
    <label for="name">Name : </label>
    <input type="text" name="name" id="name" formControlName="nameControl" required>
</div>
<div *ngIf="nameControl()?.invalid && (nameControl()?.dirty || nameControl()?.touched)">
    <div *ngIf="nameControl()?.errors?.required">Le nom est obligatoire</div>
    <div *ngIf="nameControl()?.errors?.['minlength']">Longueur du nom incorrect</div>
</div>
```



```
nameControl(): AbstractControl | null{
  return this.produitForm.get('nameControl');
}
```

Template reactive form

Name :
Le nom est obligatoire
Category : Légumes
Submit

Name : to
Longueur du nom incorrect
Category : Légumes
Submit

ANGULAR : REACTIVE FORM

Name :
Le nom est obligatoire
nom incorrect
Category :
Submit

Fonctions de validation

→ Possibilité de créer ses propres fonctions de validation.

```
checkNameValidator(control: FormControl): object | null{
  const str: string = control.value;
  if (str[0] >= 'A' && str[0] <= 'Z') {
    return null;
  } else {
    return { checkNameValidator: 'Nom non valide' };
  }
}
```



```
produitForm = new FormGroup({
  nameControl: new FormControl('', [Validators.required, Validators.minLength(3), this.checkNameValidator]),
  categoryControl: new FormControl(this.categories[0])
});
```

```
<div *ngIf="nameControl()?.errors?.['checkNameValidator']">nom incorrect</div>
```

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Rooting
- Rooting
- Forms
- **HTTP**

ANGULAR : HTTP

→ Angular propose d’utiliser des services « bas niveau » afin de procéder à un échange de donnée avec un service web côté back.

→ Utilisation du **module HTTP** (**HttpModule** et depuis la V5 **HttpClientModule**) facilitant la réalisation de requêtes http via les classes suivantes :
HttpClient, HttpHeaders, HttpInterceptor, HttpRequest, etc.

→ Permet d’invoquer des services web via les différentes méthodes HTTP :
GET, POST, PUT, DELETE

ANGULAR : HTTP – JSON SERVER

→ Pour pouvoir réaliser une démonstration, nous allons **créer un serveur** afin de pouvoir transférer des données.
Utilisation d’un serveur JSON

→ **json-server** permet d’imiter une API et de fournir un accès dynamique aux données.

→ Possibilité de lire, ajouter, mettre à jour et supprimer des données.
(GET, POST, PUT, DELETE).

→ Open-source

→ Utilise le port 3000 par défaut

ANGULAR : HTTP – JSON SERVER

Json-serveur

→ Comment l'installer ?

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

→ Création de notre fichier **db.json** (possibilité d'utiliser un générateur aléatoire de json)


```
{
    "personnes":[
    {
      "index": 0,
      "age": 31,
      "nom": "Cervantes",
      "prenom": "Mullins",
      "gender": "male",
      "company": "ENORMO",
      "email": "cervantesmullins@enormo.com"
    },
```

ANGULAR : HTTP – JSON SERVER

Json-serveur

→ Lancement du serveur

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

URL utilisée par le client pour réaliser des requêtes HTTP

```
json-server -p 5555 db.json

 \{^_^}/ hi!

 Loading db.json
 Done

 Resources
 http://localhost:5555/personnes

 Home
 http://localhost:5555

 Type s + enter at any time to create a snapshot of the database
```

ANGULAR : HTTP – JSON SERVER

Json-serveur

→ Lancement du serveur

localhost:5555/personnes

localhost:5555/personnes


```
[
  {
    "index": 0,
    "age": 31,
    "nom": "Cervantes",
    "prenom": "Mullins",
    "gender": "male",
    "company": "ENORMO",
    "email": "cervantesmullins@enormo.com"
  },
  {
    "index": 1,
    "age": 35,
    "nom": "Evans",
    "prenom": "Irwin",
    "gender": "male",
    "company": "PHOTOBIN",
    "email": "evansirwin@photobin.com"
  },
  {
```

ANGULAR : HTTP – JSON SERVER

→ **Différentes requêtes HTTP possible** :

- Pour récupérer la liste de toutes les personnes :

```
GET http://localhost:5555/personnes
```

- Pour récupérer une personne selon un identifiant :

```
GET http://localhost:5555/personnes/2
```

- Pour ajouter une nouvelle personne :

```
POST http://localhost:5555/personnes/32
```

- Pour modifier les valeurs d'une personnes :

```
PUT http://localhost:5555/personnes/2
```

- Pour supprimer une personne :

```
DELETE http://localhost:5555/personnes/2
```

ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

Dans le **module principale** de l'application :

```
import { HttpClientModule } from '@angular/common/http';
```

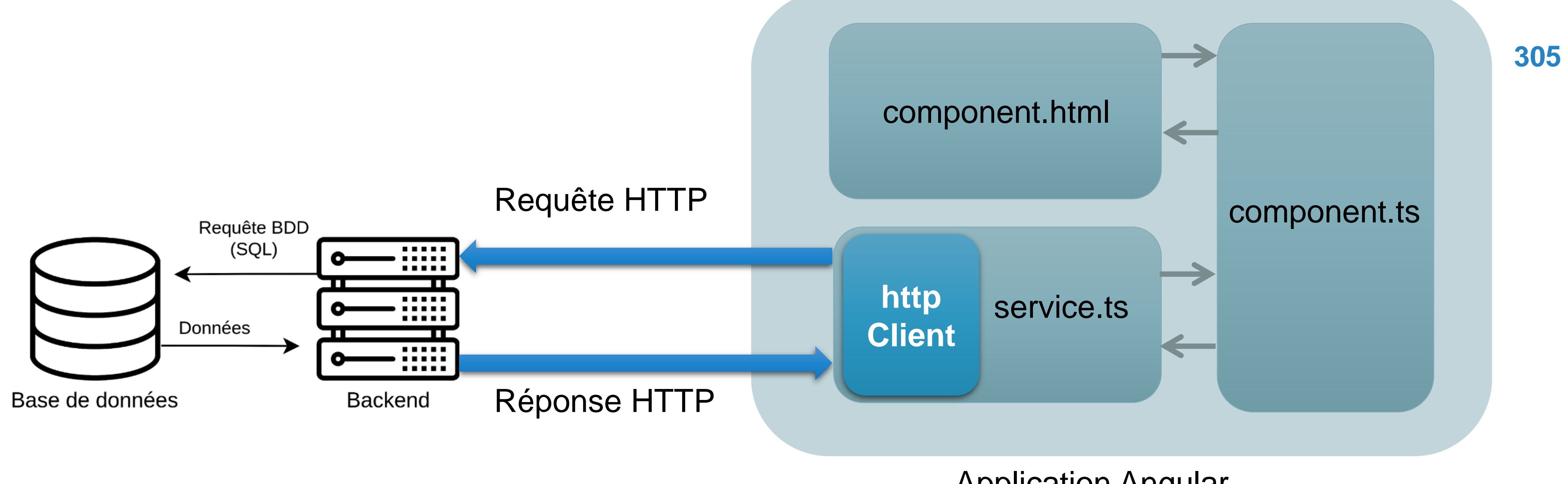


```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
```

ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

Utilisation d'un **service** pour gérer les transferts de données vers un serveur :



ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

- Les données sont saisies dans le **template** d'un composant (*component.html*)
- La **classe du composant** (*component.ts*) peut récupérer les données du template pour les passer au service (ou inversement récupérer du service pour les envoyer au template).
- Grâce à **l'injection de dépendance du service** (*service.ts*) dans la classe du composant, ce dernier peut l'utiliser pour persister ou récupérer des données.
- En faisant une injection de dépendance de la classe **HttpClient** dans le service, ce dernier peut effectuer des requêtes HTTP en précisant chaque fois la méthode et l'URL.

ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

Utilisation d'un **service** pour gérer les transferts de données vers un serveur :

exempleHttp > src > app > TS personne.service.ts > ...

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

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

  url = "http://localhost:3000/personnes";

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

URL de base pour les requêtes HTTP

Injection du service HTTP dans le service Personnage

ANGULAR : HTTP

→ **Comment utiliser le module HTTP** ?

Dans notre nouveau composant **Personne** : Injection du service **PersoneService**.


```
 9  export class PersonneComponent implements OnInit {
10
11    constructor(private personneService: PersonneService) { }
12
13    ngOnInit(): void {
14        //récupération des données du serveur
15    }
16  }
```

Fichier personne.component.ts

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

→ Avant tout, nous avons créé une interface **IData** contenant toutes les caractéristiques de notre personnage (indépendant du contenu du fichier json) :

exempleHttp > src > app > TS IData.ts > ...

```
1  export interface IData {
2      index: number;
3      age: number;
4      nom: string;
5      prenom: string;
6      gender: string;
7      company: string;
8      email: string;
9  }
```

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

→ Et ajouter à notre composant **Personne** afin de pouvoir visualiser les données récupérées du serveur :

exempleHttp > src > app > personne > TS personne.component.ts > ...

```
10  export class PersonneComponent implements OnInit {
11
12    personnes: IData[] = [];
```

exempleHttp > src > app > personne > <> personne.component.html >

```
1  <h2>Liste des personnes :</h2>
2  <ul>
3      <li *ngFor="let perso of personnes">
4          {{ perso.prenom }} {{ perso.nom }}
5      </li>
6  </ul>
```

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

→ Utilisation de la méthode **HttpClient.get()**.

→ Utilise les **Observables** (rxjs). *(revoir la partie sur les services pour le rappel)*

exempleHttp > src > app > TS personne.service.ts > ...

```
11  export class PersonneService {
12
13    url = "http://localhost:3000/personnes";
14
15    constructor(private http: HttpClient) { }
16
17    getAll(): Observable<IData[]>{
18      return this.http.get<IData[]>(this.url);
19    }
20  }
```

La réponse de l'appel HTTP est un observable non typé par défaut.

Observable de IData[]
(attention le serveur peut envoyer autre un autre type de données)

Endpoint URL

ANGULAR : HTTP

Pas besoin de transformer les données reçues du serveur dans cet exemple

1) Récupération de toutes les données Personne :

exempleHttp > src > app > personne > TS personne.component.ts > ...

```
10  export class PersonneComponent implements OnInit {
11
12    personnes: IData[] = [];
13
14    constructor(private personneService: PersonneService) { }
15
16    ngOnInit(): void {
17      //récupération des données du serveur
18      this.personneService.getAll().subscribe( res => {
19        this.personnes = res;
20      })
21    }
22  }
```

Res est de type IData[]

```
(res:IData[])
```

Souscription au retour de la méthode *getAll()* du service

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

Résultat obtenu :

Liste des personnes :

- Mullins Cervantes
- Irwin Evans
- Wagner Eunice
- Sykes Bean
- Robertson Luisa
- Case Byers

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

Comment faire si la réponse de notre requête ne correspond pas à nos structures de données ?

→ Nouvelle **interface** :

```
exempleHttp > src > app > TS IDataLight.ts > ...
1  export interface IDataLight {
2      index: number;
3      age: number;
4      lastname: string;    //dans IData : nom
5      firstname: string;  //dans IData : prenom
6  }
```

→ Modification du composant **Personne** pour utiliser cette interface :

```
exempleHttp > src > app > personne > TS personne.component.ts > ...
11  export class PersonneComponent implements OnInit {
12
13    personnes: IData[] = [];
14    personnesBis: IDataLight[] = [];
```

```
exempleHttp > src > app > personne > <> personne.component.html > ...
7   <h2>Liste des personnes IDATALIGHT:</h2>
8   <ul>
9       <li *ngFor="let perso of personnesBis">
10          {{ perso.firstname }} {{ perso.lastname }}
11      </li>
12  </ul>
```

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

→ On souhaite avoir une méthode `get()` retournant un `Observable<IDataLight[]>`

→ **Problème :**

```
getAllBis(): Observable<IDataLight[]>{
  return this.http.get<IData[]>(this.url);
}
```

→ Utilisation de l'opérateur **map** de **RxJS** pour transformer la réponse (avec l'async **pipe**)

```
getAllBis(): Observable<IDataLight[]>{
  return this.http.get<IData[]>(this.url).pipe(
    map( res => res.map( data => {
      return {
        index: data.index,
        age: data.age,
        lastname: data.nom,
        firstname: data.prenom
      } as IDataLight
    }))
  );
}
```

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

Possibilité d'ajouter diverses options à la méthode **HttpClient.get()**.

```
options: {
    ...
    observe?: 'body' | 'events' | 'response',
    ...
    responseType?: 'arraybuffer'|'blob'|'json'|'text',
    ...
  }
```

Nous voulons la totalité de la réponse à la requête http


```
getAll(): Observable<IData[]>{
  return this.http.get<IData[]>(this.url, { responseType: 'json' });
}
```

Par défaut : body et json

ANGULAR : HTTP

Headers

- cache-control: no-cache
- content-type: application/json; charset=utf-8
- expires: -1
- pragma: no-cache

1) Récupération de toutes les données Personne :

Réponse entière ?

exempleHttp > src > app > TS personne.service.ts > ...

```
35  getAllResponse(): Observable<HttpResponse<IData[]>>{
36    return this.http.get<IData[]>(this.url, {observe: 'response'});
37  }
```

exempleHttp > src > app > personne > TS personne.component.ts > ...

```
32  showAllResponse() {
33    this.personneService.getAllResponse().subscribe( res => {
34      //headers (string[])
35      const keys = res.headers.keys();
36      this.headers = keys.map(key =>
37        `${key}: ${res.headers.get(key)}`);
38      //body (IData[])
39      this.personnes = res.body!;
40    })
41  }
42 }
```

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

Possibilité d'avoir des **erreurs** lors des requêtes HTTP

→ Le serveur peut rejeter la requête HTTP (code de retour 404 ou 500 par exemple) : *error response*

→ Ou alors problème côté client (problème de réseau, exception déclenchée par un operateur RxJS, etc.). Les erreurs ont pour statut 0.

ANGULAR : HTTP

Exemple de gestion d’erreur :

exempleHttp > src > app > TS personne.service.ts > PersonneService

```
18  getAll(): Observable<IData[]>{
19    return this.http.get<IData[]>(this.url).pipe(
20      catchError(this.handleError)
21    );
22  }
```

exempleHttp > src > app > TS personne.service.ts > PersonneService

```
41  private handleError(error: HttpErrorResponse) {
42    if (error.status === 0) {
43      // A client-side or network error occurred. Handle it accordingly.
44      console.error('An error occurred:', error.error);
45    } else {
46      // The backend returned an unsuccessful response code.
47      // The response body may contain clues as to what went wrong.
48      console.error(
49        `Backend returned code ${error.status}, body was: `, error.error);
50    }
51    // Return an observable with a user-facing error message.
52    return throwError(
53      'Something bad happened; please try again later.');
54  }
```


https://angular.io

ANGULAR : HTTP

1) Récupération de toutes les données Personne :

RxJS permet également de refaire une requête http si celle-ci a échouée.

→ `retry()` permet de souscrire une nouvelle fois à un Observable.


```
getAll(): Observable<IData[]>{
  return this.http.get<IData[]>(this.url).pipe(
    retry(3),
    catchError(this.handleError)
  );
}
```

ANGULAR : HTTP

2) Ajout d’une personne dans le fichier json :

Comment ajouter une nouvelle personne dans le fichier *db.json* ?

→ Utilisation de la méthode `HttpClient.post()`

Fonctionne de manière similaire à `get()`

Possibilité de spécifier un header

Attention, le fichier db.json doit posséder un attribut « id » (automatiquement rempli avec post)

→ Les données de la personne peuvent être récupérée à partir d’un formulaire.

exempleHttp > src > app > TS personne.service.ts > ...

```
61  addPersonne(personne: IData): Observable<IData> {
62    return this.http.post<IData>(this.url, personne);
63  }
```

ANGULAR : HTTP

2) Ajout d'une personne dans le fichier json :

Comment ajouter une nouvelle personne dans le fichier *db.json* ?

```
exempleHttp > src > app > personne > TS personne.component.ts > ...
55    addPersonne() {
56      this.personneService.addPersonne(this.perso).subscribe( res => {
57          console.log(res);
58          this.personnes.push(res);
59      })
60    }
```

→ Possibilité d'ajouter une gestion d'erreur comme pour `get()`

ANGULAR : HTTP

2) Ajout d'une personne dans le fichier json :

Comment ajouter une nouvelle personne dans le fichier *db.json* ?

Résultat :

Liste des personnes IDATA:

- Mullins Cervantes
- Irwin Evans
- Wagner Eunice
- Sykes Bean
- Robertson Luisa
- Case Byers
- Titi Toto


```
▶ Object { id: 7, age: 55, nom: "Toto", prenom: "Titi", gender: "No", company: "Esir", email: "Esir@Esir.com" }
```

ANGULAR : HTTP

Remarque :

→ Possibilité d'utiliser le package **concurrently** pour ne plus à avoir démarrer séparément les deux serveurs Angular et json-server.

```
npm install concurrently --save
```


→ Package NodeJS permettant d'exécuter plusieurs commandes simultanément.

```
npm start
```

→ `concurrently "command1 arg" "command2 arg"`

```
exempleHttp > {} package.json > {} dependencies
4    "scripts": {
5      "ng": "ng",
6      "start": "concurrently \"ng serve\"  \"json-server db.json\" ",
7      "build": "ng build"
```

ANGULAR : HTTP

→ Les ressources REST ont besoin d'un authentification et d'une autorisation.

→ JSON web tokens (JWT)
Généré par un web service et aide à la communication entre le client et le serveur.

→ Création d'un service pour enregistrer le **token** et le réutiliser.

→ Mise en place d'un **intercepteur** pour injecter des headers dans tous les requêtes d'authentification (et ne pas à avoir répéter du code)

→ Ajout d'une **garde** pour restreindre l'accès à certains composants aux personnes identifiés uniquement.

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Rooting
- Rooting
- Forms
- HTTP
- **Pipes**

ANGULAR : PIPES

→ Les données obtenues peuvent ne pas être affichées de la manière souhaitée dans la vue.
Par exemple : la date, la monnaie, l'ordre d'une liste, etc.

→ La solution avec Angular : les **pipes**.
Fonctions prenant en entrée une valeur et retournant la valeur transformée.

→ Plusieurs fonctions existent déjà : `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, `DecimalPope`, `PercentPipe` etc.

ANGULAR : PIPES

→ Exemple pour les dates :

```
today = new Date();
```

```
<p>Nous sommes le : {{ today | date }}</p>
<p>Nous sommes le : {{ today | date:"fullDate" }}</p>
<p>Nous sommes le : {{ today | date:"MM/dd/yyyy" }}</p>
```

Nous sommes le : Jan 7, 2022

Nous sommes le : Friday, January 7, 2022

Nous sommes le : 01/07/2022

→ Utilisation de la fonction `registerLocalData`

```
exempleForm > src > app > TS app.module.ts > AppModule
14  import { FormsModule } from '@angular/forms';
15  import { ReactiveFormsModule } from '@angular/forms';
16  import { registerLocaleData } from '@angular/common';
```

```
providers: [{provide: LOCALE_ID, useValue: "fr-FR"}],
```

Nous sommes le : 7 janv. 2022

Nous sommes le : vendredi 7 janvier 2022

Nous sommes le : 01/07/2022

ANGULAR : PIPES

→ Possibilité de créer son propre pipe :

exempleForm > src > app > TS greeting.pipe.ts > ...

```typescript
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'greeting'
})
export class GreetingPipe implements PipeTransform {

  transform(value: string, gender: string): string {
    if(gender === "M"){
      return `Bonjour monsieur ${value}`;
    } else if(gender === "F"){
      return `Bonjour madame ${value}`;
    } else {
      return `Bonjour ${value}`;
    }
  }

}
```

exempleForm > src > app > TS app.module.ts > ...

```typescript
import { GreetingPipe } from './greeting.pipe';

@NgModule({
  declarations: [
    AppComponent,
    ProductsComponent,
    GreetingPipe
  ],
```

```html
<p>{{ name | greeting:"M" }}</p>
<p>{{ name | greeting:"" }}</p>
```

Bonjour monsieur John Doe

Bonjour John Doe

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

- Component / Template
- Data Binding
- Component Interaction
- Service
- Rooting
- Rooting
- Forms
- HTTP
- Pipes
- **Directives**

ANGULAR : DIRECTIVES

Au cours des différents exemple du cours, deux types de directives Angular ont été utilisées :

→ **Directives structurelles** : modification de l'arborescence du DOM.
ngIf, ngFor, NgSwitch, etc.

```
<div *ngIf="condition">Hello World</div>
<ul>
    <li *ngFor="let item of ['un', 'deux', 'trois']">{{item}}</li>
</ul>
```

→ **Directives d'attributs** : pas de modification du DOM, mais des attributs et propriétés des balises HTML existantes.
ngStyle, ngClass, ngModel, etc.

```
<div [ngStyle]="{color: 'blue'}">Hello World</div>
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

ANGULAR : DIRECTIVES

Possibilité de créer ses propres directives.

Exemple pour un directive d’attribut : Color.

On souhaite modifier la couleur de fond d’un élément s’il est survolé.

exempleForm > src > app > TS color.directive.ts > ...

```
1   import { Directive } from '@angular/core';
2
3   @Directive({
4     selector: '[appColor]'
5   })
6   export class ColorDirective {
7
8     constructor() {  }
9
10  }
```

```
import { ColorDirective } from './color.directive';

@NgModule({
  declarations: [
    AppComponent,
    FormTDComponent,
    FormRComponent,
    ProductsComponent,
    GreetingPipe,
    ColorDirective
  ],
```

A intégrer dans le app module

ANGULAR : DIRECTIVES

Possibilité de créer ses propres directives.

Injection de dépendance de `ElementRef` pour référencer les éléments concernés par la directive.

exempleForm > src > app > TS color.directive.ts > ...

```
1   import { Directive, ElementRef } from '@angular/core';
2
3   @Directive({
4     selector: '[appColor]'
5   })
6   export class ColorDirective {
7
8     constructor(private el: ElementRef) {
9         el.nativeElement.style.background = 'red';
10    }
11
12  }
```

```
<p appColor>Hello world</p>
```

Hello world

ANGULAR : DIRECTIVES

Possibilité de créer ses propres directives.

Utilisation du décorateur `@HostListener` pour rattacher le changement de couleur à un évènement.

Hello world

exempleForm > src > app > TS color.directive.ts > ...

```
import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({
  selector: '[appColor]'
})
export class ColorDirective {

  constructor(private el: ElementRef) {  }

  @HostListener('mouseenter') onMouseEnter(): void {
    this.changerCouleur('red');
  }
  @HostListener('mouseleave') onMouseLeave(): void {
    this.changerCouleur('white');
  }
  changerCouleur(couleur: string){
    this.el.nativeElement.style.background = couleur;
  }
}
```

ANGULAR : DIRECTIVES

Possibilité de créer ses propres directives.

Utilisation du décorateur `@Input` pour que la couleur soit un paramètre de l'attribut `appColor`.

```
<p appColor="blue">Hello world</p>
```

Hello world

```
export class ColorDirective {

  @Input('appColor') couleur = '';

  constructor(private el: ElementRef) {  }

  @HostListener('mouseenter') onMouseEnter(): void {
    this.changerCouleur(this.couleur);
  }
  @HostListener('mouseleave') onMouseLeave(): void {
    this.changerCouleur('white');
  }
  changerCouleur(couleur: string){
    this.el.nativeElement.style.background = couleur;
  }
}
```

FRAMEWORK ANGULAR

Utilisation de Framework

Notion de composant web

Concept d'Angular

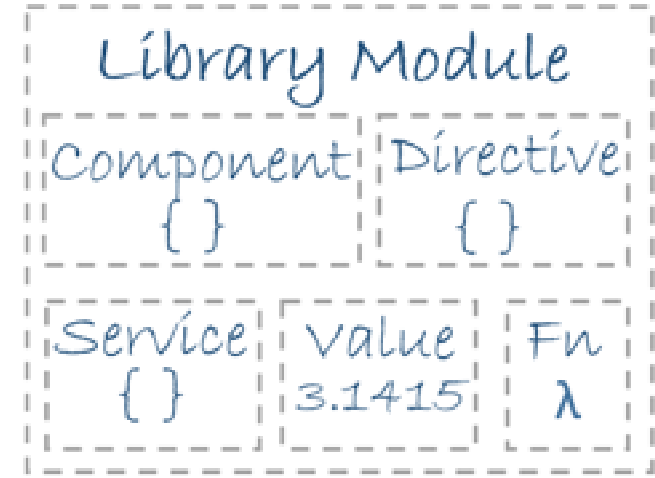
- Component / Template
- Data Binding
- Component Interaction
- Service
- Rooting
- Rooting Forms
- HTTP
- Pipes
- Directives
- **Module**

ANGULAR : MODULES

→ Un module correspond à une partie de l'application que l'on veut importer ou exporter.

Décorateur `@ngModule` avec plusieurs propriétés possibles : `declarations`, `exports`, `imports`, `providers`, `bootsrap` (pour le root module uniquement).

→ Possibilité de créer des sous modules dans une application.



ANGULAR : SOUS-MODULES

→ Nouveau sous module `Fruits` dans notre application.

→ Nouveau composant `Pomme`

```
exempleForm > src > app > modules > fruits > TS fruits.module.ts > ...
1   import { NgModule } from '@angular/core';
2   import { CommonModule } from '@angular/common';
3
4   import { FruitsRoutingModule } from './fruits-routing.module';
5   import { PommeComponent } from './pomme/pomme.component';
6
7   @NgModule({
8     declarations: [
9       PommeComponent
10    ],
11    imports: [
12      CommonModule,
13      FruitsRoutingModule
14    ]
15  })
16  export class FruitsModule { }
```

Module contenant les pipes et les directives

ANGULAR : SOUS-MODULES

→ Besoin d'importer ce nouveau module dans le module principal de notre application.

exempleForm > src > app > TS app.module.ts > AppModule

```
19  import { FruitsModule } from './modules/fruits/fruits.module';
20
21  @NgModule({
22    declarations: [
23      AppComponent,
24      FormTDComponent,
25      FormRComponent,
26      ProductsComponent,
27      GreetingPipe,
28      ColorDirective
29    ],
30    imports: [
31      BrowserModule,
32      AppRoutingModule,
33      FormsModule,
34      ReactiveFormsModule,
35      FruitsModule
36    ],
```

ANGULAR : SOUS-MODULES

→ Possibilité d'avoir un module de routage pour ce sous-module.

→ **Eager loading** vs lazy loading

→ Dans le app routing module

```
form > src > app > TS app-routing.module.ts > [@] routes
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { FormRComponent } from './form-r/form-r.component';
import { FormTDComponent } from './form-td/form-td.component';
import { PommeComponent } from './modules/fruits/pomme/pomme.component';

const routes: Routes = [
  {path:'formtd', component:FormTDComponent},
  {path:'formr', component:FormRComponent},
  {
    path:'fruits', children: [
      { path:'pomme', component:PommeComponent}
    ]
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
```

ANGULAR : SOUS-MODULES

→ Possibilité d'avoir un module de routage pour ce sous-module.

→ **Eager loading** vs lazy loading

→ Ou dans le rooting module du nouveau module.

exempleForm > src > app > modules > fruits > TS fruits-routing.module.ts > ...

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { PommeComponent } from './pomme/pomme.component';

const routes: Routes = [
  { path:'pomme', component:PommeComponent}
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class FruitsRoutingModule { }
```

ANGULAR : SOUS-MODULES

→ Possibilité d'avoir un module de routage pour ce sous-module.

→ Eager loading vs **lazy loading**

→ Utilisation de `loadChildren` et des promesses.

exempleForm > src > app > TS app-routing.module.ts > AppRoutingModule

```
6  const routes: Routes = [
7    {
8      path:'fruits',
9      loadChildren: () => import('./modules/fruits/fruits.module')
10       .then(m => m.FruitsModule)
11   }
12 ];
```

exempleForm > src > app > modules > fruits > TS fruits-routing.module.ts >

```
5  const routes: Routes = [
6    { path:'pomme', component:PommeComponent}
7  ];
```

+ Suppression du module `FruitsModule` dans les imports du `app` module.

ANGULAR : MODULES

→ Lors de l'apprentissage d'Angular : utilisation de module et moins création de module.

→ Plusieurs modules sont déjà créés :

- Angular material (https://material.angular.io/)
- NGX-Bootstrap (https://valor-software.com/ngx-bootstrap)
- primeNG (https://www.primefaces.org/primeng/)
- Etc.

ANGULAR