FORMATION ANGULAR 11

ANTHONY DI PERSIO

TABLE DES MATIÈRES

01

DECOUVERTE D'ANGULAR

Découverte du Framework Angular et installation de l'environnement

DECOUVERTE D'ANGULAR

Angular est un **Framework** open-source **Front-End**

- **Développé** en partie par **Google** et d'autres **sociétés partenaires**.
- Totalement **orienté interface Utilisateur** (UI)
- Basé sur **TypeScript**

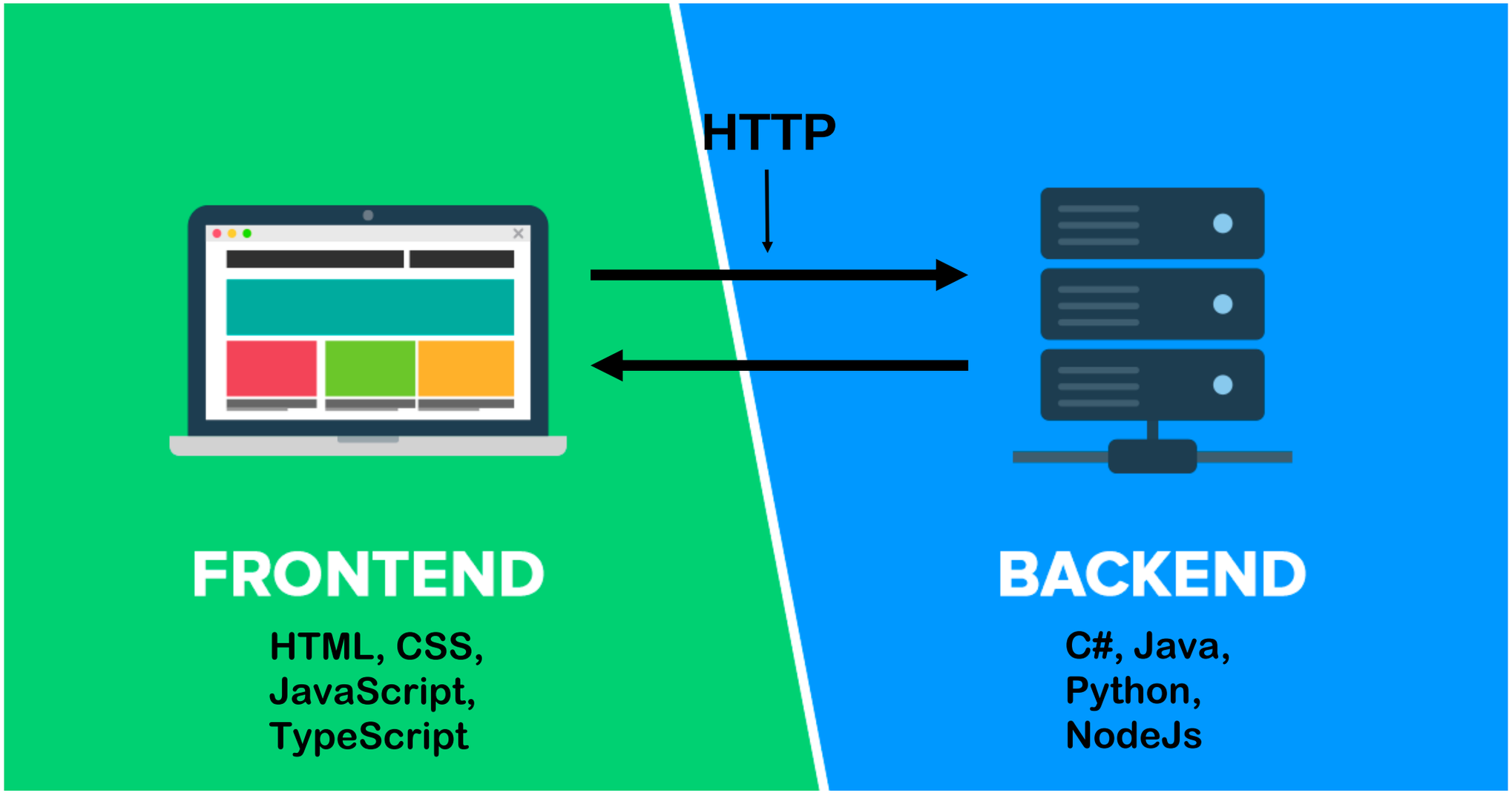
DECOUVERTE D'ANGULAR

Angular permet de créer des interfaces Web dynamique

- Donne à l'application une structure propre
- Permet la réutilisation du code et des composants
- Facilite la maintenabilité
- Permet la réalisation de tests unitaires

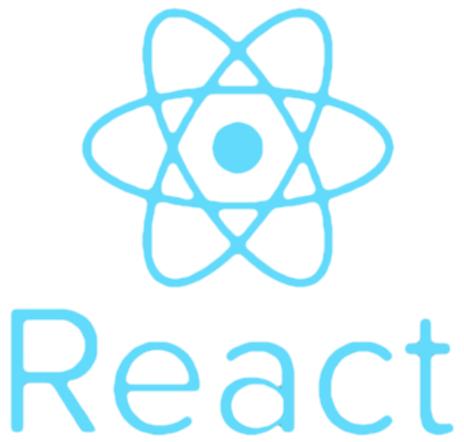
DECOUVERTE D'ANGULAR

Angular s'inscrit dans l'architecture Front-End, Back-End



DECOUVERTE D'ANGULAR

Angular fait partie des principaux Framework Front actuels



DECOUVERTE D'ANGULAR

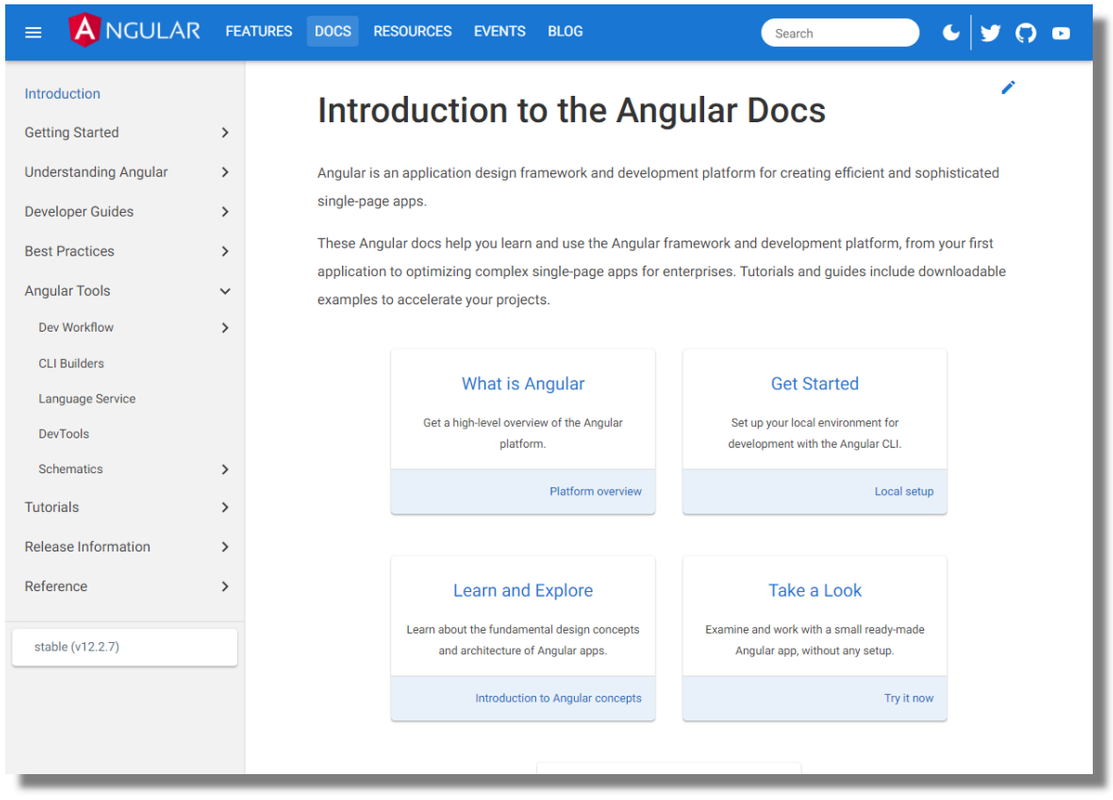
Angular utilise le bundle webpack

- Afin de compiler les fichiers TypeScript en JavaScript
  - Assure la compatibilité JavaScript ECMA6
  - Garanti la compatibilité avec la majorité des browsers
- Permet de compiler deux environnements
  - Mode développement (**J**ust-**I**n-**T**ime)
  - Mode production (**A**head-**O**f-**T**ime)

DECOUVERTE D'ANGULAR

Documentation sur Angular

- Documentation officielle
  - **Angular:** https://angular.io/docs



DECOUVERTE D'ANGULAR

Installation de l'environnement de développement Angular

- Angular nécessite **node.JS** pour fonctionner
  - Node JS : https://nodejs.org
- Utilisation du **N**ode **P**ackage **M**anager pour installer **Angular CLI**
  - `$ npm install -g @angular/cli`
- Installation de l'IDE Visual code de Microsoft
  - https://code.visualstudio.com

DECOUVERTE D'ANGULAR

Création de notre premier projet Angular avec la CLI

- Commande pour créer un nouveau projet
  - `$ ng new hello-world`
- Se positionner à la racine de notre projet
  - `$ cd hello-world`
- Compiler et démarrer le serveur
  - `$ ng serve`

02

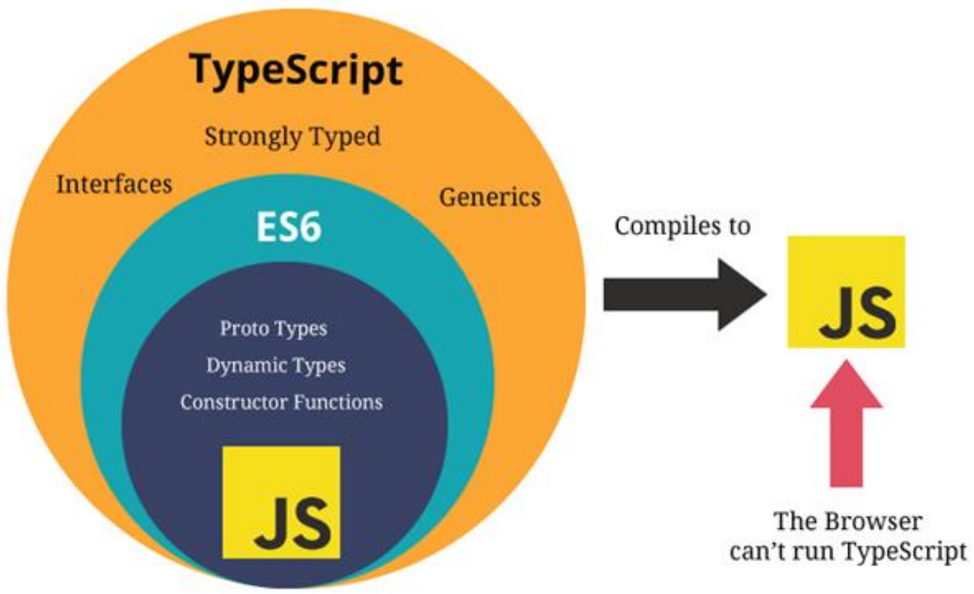
LE TYPESCRIPT

Tour d’horizon des spécificités du TypeScript

LES CONVENTIONS DU TYPESCRIPT

La découverte du langage TypeScript

- Le TypeScript est un langage de programmation développé par Microsoft en 2012
  - Il est open-source
  - TypeScript est un « Super » JavaScript



LES CONVENTIONS DU TYPESCRIPT

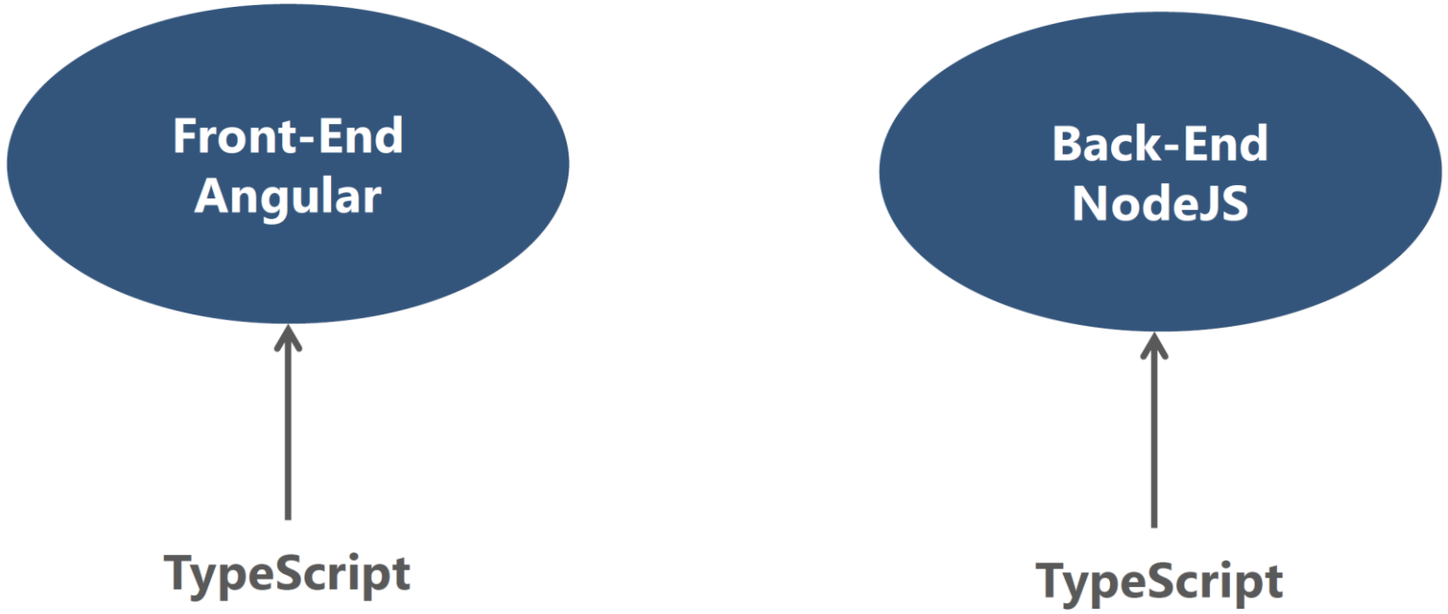
La découverte du langage TypeScript

- Les avantages du TypeScript par rapport au JavaScript
  - Le typage des variables
  - Il est orienté objet
  - Il permet de capturer les erreurs durant la compilation
    - ✓ Compile-time-errors

LES CONVENTIONS DU TYPESCRIPT

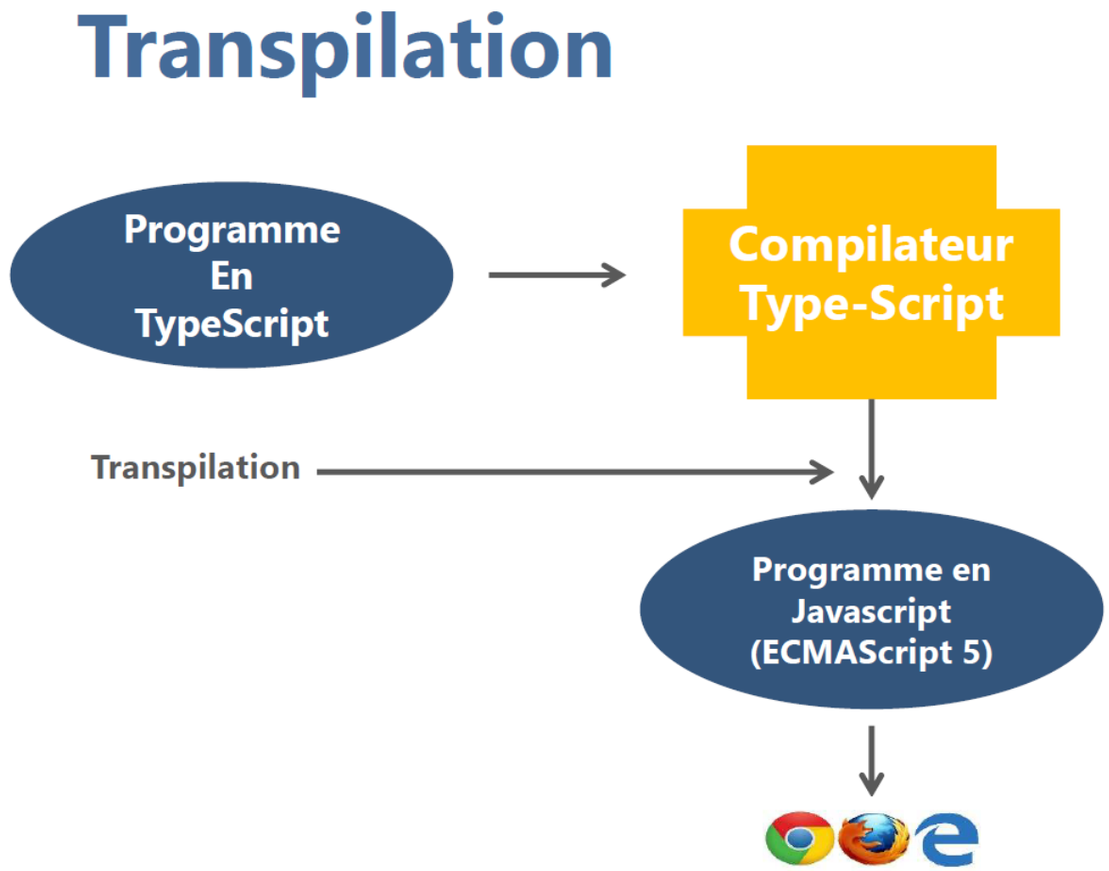
L’utilisation de TypeScript dans les projets

- On utilise le TypeScript pour le Front comme le Back-End



LES CONVENTIONS DU TYPESCRIPT

- Afin de rendre compatible le TypeScript avec les navigateur
  - Les code est **transpillé**



LES CONVENTIONS DU TYPESCRIPT

- Les différentes versions de JavaScript

ES5 = ES2009

ES6 = ES2015 (Tous les navigateurs modernes supportent l'ES6,Angular, ViewJS, React)

ES7 = ES2016

ES8 = ES2017

ES9 = ES2018

ES10 = ES2019

ES11 = ES2020

LES CONVENTIONS DU TYPESCRIPT

- Afin de commenter le code en TypeScript, deux type d'annotation de commentaires peuvent êtres employés

  - Le commentaire de ligne

    - ✓ // Le reste de la ligne est commenté

  - Le commentaire multilignes

    - ✓ /**
      * Tout le texte situé entres les deux délimiteurs
      */

LES CONVENTIONS DU TYPESCRIPT

- Les **Types** en **TypesScript**

  - Déclaration d'une **variable** en **TypeScript**

    - ✓ **var** variable : Type;
    - ✓ **let** variable : Type;
    - ✓ **const** variable : Type;

  - **Type**:any, number, string, boolean, enum, array, void…

    - ✓ **Let** code : **Number**=1357;
    - ✓ **Let** nom : **string**=« Jeanne »;

LES CONVENTIONS DU TYPESCRIPT

- Les **opérateurs** du **TypeScript**
  - Les **opérateurs arithmétiques**

Opérateur	Fonction
+	Addition
-	Soustraction
/	Division
*****	Multiplication
%	Modulo (reste de la division Euclidienne)

  - Les **opérateurs** Pré-Post **Incrémentation** et autres

Opérateur	Fonction
++	Incrémente de 1
--	Décrémente de 1
+=	Addition (à une variable)
-=	Soustraction (à une variable)
/=	Division (à une variable)
***=**	Multiplication (à une variable)

LES CONVENTIONS DU TYPESCRIPT

- Les **classes** en **TypeScript** se composent

  - D'attributs

    - ✓ Class Personne
      {
      Nom : string;
      Prenom : string;
      Age : Number;
      }

LES CONVENTIONS DU TYPESCRIPT

- Les **classes** en **TypeScript** se composent

  - De méthodes …

    - ✓ **AffichageInfo()**

```
{
    console.log(« Nom  : » + Nom + « - Prénom : » +
        Prenom + « - Age  : » + Age);
}
```

LES CONVENTIONS DU TYPESCRIPT

- Les **classes** en **TypeScript** se composent
  - D'un constructeur permettant d'initialiser les attributs
    - ✓

```
Class Personne
{
    Nom:String;
    Prenom : String;
    Age : Number;
    constructor(nom:string, prenom:string, age:number)
    {
        this.Nom = nom;
        this.Prenom = prenom;
        this.Age = age;
    }
}
```

LES CONVENTIONS DU TYPESCRIPT

- Une fois la classe complétée

```
Class Personne
{
        Nom:String;
        Prenom : String;
        Age : Number;
        constructor(nom:string, prenom:string, age:number)
        {
            this.Nom = nom;
            this.Prenom = prenom;
            this.Age = age;
        }
        AffichageInfo()
        {
            console.log(« Nom  : » + Nom + « - Prénom : » +
                        Prenom + « - Age  : » + Age);
        }
}
```

LES CONVENTIONS DU TYPESCRIPT

- Nous pouvons procéder à **l'instanciation** de celle-ci

  - Création d'une instance de **Personne**

    - ✓ **let** personne1:**Personne=new Personne();**

```
personne1.Nom="Toto";
personne1.Prenom="Titi";
personne1.Age=29;
personne1.AffichageInfo();
```

LES CONVENTIONS DU TYPESCRIPT

La notion **d'héritage** en **TypeScript**

- L'héritage permet de créer une nouvelle classe à partir d'une classe existante

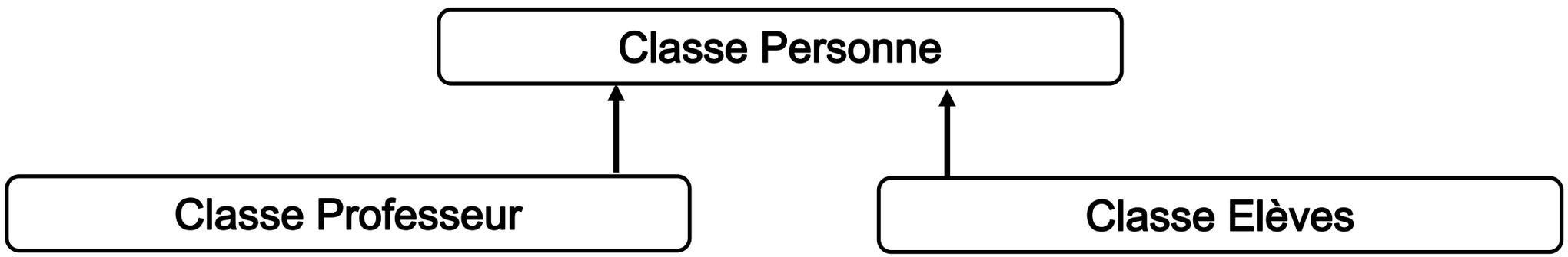
  - La classe enfant hérite donc des attributs et méthodes de la classe mère

  - La classe enfant a la possibilité de se « spécifier » de d'y ajouter ses propres attributs et méthodes

    - ✓ Possibilité **d'override** les méthodes de la classe mère pour les spécifier

LES CONVENTIONS DU TYPESCRIPT

La notion **d'héritage** en TypeScript



- La classe Etudiant hérite de la classe personne
- La classe Professeur hérite de la classe personne

LES CONVENTIONS DU TYPESCRIPT

La notion **d'héritage** en **TypeScript**

- Implémenter l'héritage en TypeScript

```
Class Personne
{

}

Class Etudiant extends Personne
{

}
```

- La classe **Etudiant** hérite de la classe **Personne**

LES CONVENTIONS DU TYPESCRIPT

La notion **d'héritage** en **TypeScript**

- Implémenter l'héritage en TypeScript

```
Class Personne
{
        constructor (public Nom:string, public Prenom:string, public Age:number)
        {
        }
}

Class Etudiant extends Personne
{
        constructor(Nom:string,Prenom:string,Age:number,public NumeroCarte: number)
        {
                super(Nom,Prenom,Age);
        }
}
```

➢ **let** etudiant1 = new **Etudiant**("Toto","Titi",**18**, **4526**);

LES CONVENTIONS DU TYPESCRIPT

Les **Interfaces** en **TypeScript**

- En **TypeScript**, une **Classe** peut **implémenter** une ou plusieurs **interfaces**
  - Elle doit **s'engager** à **implémenter** les **propriétés** et les méthodes :

```
class Etudiant implements IPersonne
{
        Nom: string;
        Prenom: string;
        Age: number;

        Affichage(): void {
                console.log("Nom :"+this.Nom);
                console.log("Prénom :"+this.Prenom);
                console.log("Age :"+this.Age);
                }
}
```

LES CONVENTIONS DU TYPESCRIPT

Les **Interfaces** en **TypeScript**

- Nous pouvons déclarer une variable de Type **Interface**

  - Exemple: **var** personne1**:IPersonne**={
    nom:"Robin",
    prenom:"Patrick",
    age:39,
    Affichage()
    {
        return "Affichage";
    }
}

LES CONVENTIONS DU TYPESCRIPT

Les **Interfaces** en **TypeScript**

- Nous pouvons élargir une interface avec le mot clé **extends**

  - Exemple: interface **IEtudiant** **extends** **IPersonne** {
    numeroCarte?:string, //Nullable
    Annee:string,
    Branche:number,
    }

LES CONVENTIONS DU TYPESCRIPT

Le **polymorphisme** avec des **Interfaces** en **TypeScript**

- C'est un **concept** qui permet de **traiter** des **objets** de **différents types** d'une **manière identique**

  - Ici on utilise les interfaces pour obtenir un polymorphisme:

```
interface Connector
{
    doConnect(): boolean;
}
```

LES CONVENTIONS DU TYPESCRIPT

Le **polymorphisme** avec des **Interfaces** en **TypeScript**

```
interface Connector
{
	doConnect(): boolean;
}
```

```
export class WifiConnector implements
Connector
{
	public doConnect(): boolean{
		console.log("Connecting via wifi");
		return true;
	}
}
```

```
export class BluetoothConnector implements
Connector
{
	public doConnect(): boolean{
		console.log("Connecting via Bluetooth");
		return true;
	}
}
```

```
export class System {
	constructor(private connector: Connector)
	{ #inject Connector type
		connector.doConnect()
	}
}
```

03

TEMPLATES & DIRECTIVES

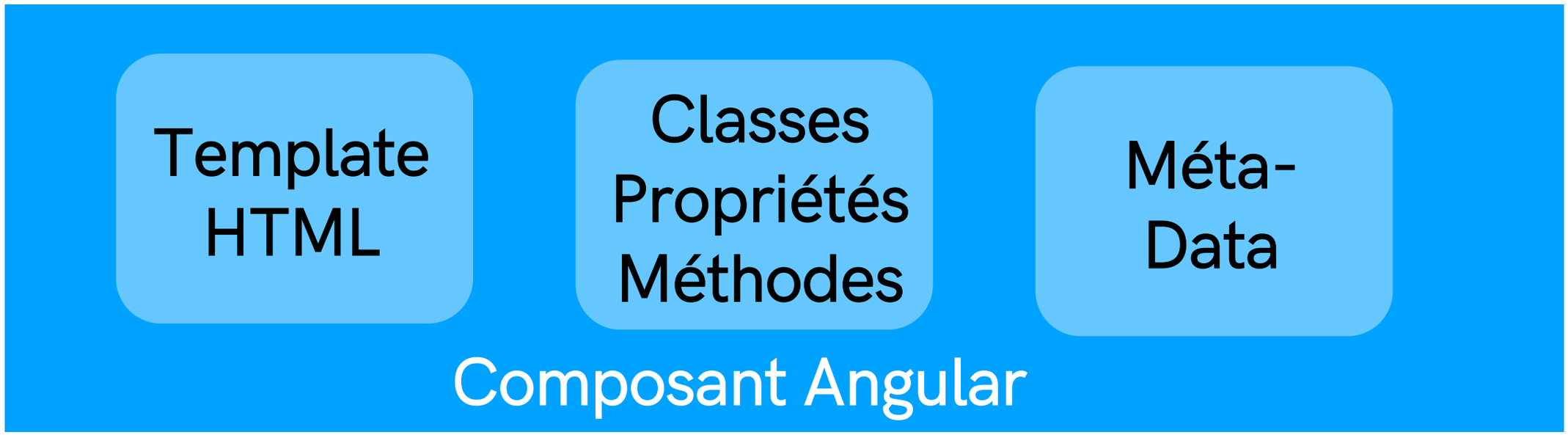
Détail des templates HTML et des directives dans Angular

TEMPLATES & DIRECTIVES

Les Templates en Angular

- Angular est orienté composants et chacun d'entre eux dispose d'une partie dédié à l'affichage dans le navigateur

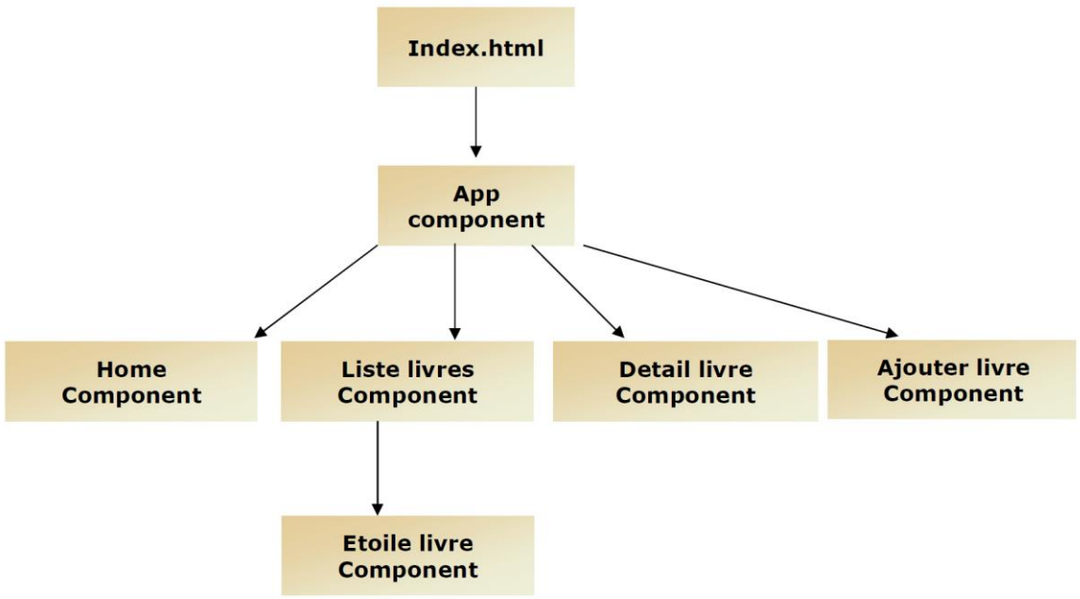
  - C'est le Template HTML



TEMPLATES & DIRECTIVES

Les Templates en Angular

- Donc chaque composant comporte un « bout » de la page HTML dans une section **template** ou un **fichier dédié**
  - ✓ Permettant de contenir le code HTML
  - ✓ Les directives implémenté par Angular



TEMPLATES & DIRECTIVES

Les **directives** structurelle **d'Angular**

- Les directives structurelles d'Angular sont les directives qui permettent de manipuler les éléments du DOM

  - **ngIf** pour les affichage conditionnel
  - **ngfor** pour les boucles itératives
  - **ngClass** pour les affectation de class CSS
  - **ngStyle** pour les affectation de style CSS

TEMPLATES & DIRECTIVES

Les **directives** structurelle **d'Angular**

- **La directive *ngIf** (pour l'application conditionnelle)

```
@Component
({
    selector:'pm-listCourses',
    template:`
    <div *ngIf="listCours.length>0">
    <h1> la liste des cours <h1/>
    </div>
    `
})
export class ListCourseComponent{
    listCours=["Cours1","Cours2","Cours3"]
}
```

TEMPLATES & DIRECTIVES

Les **directives** structurelle **d'Angular**

- **La directive *ngFor** (pour les boucles d'itération)

```
@Component
({
    selector:'pm-listCourses',
    template:`
    <div *ngFor="let cours of listCours">
        <h1> {{cours}}<h1/>
    </div>
    `
})
export class ListCourseComponent{
    listCours=["Cours1","Cours2","Cours3"]
}
```

TEMPLATES & DIRECTIVES

Les **directives** structurelle **d'Angular**

- **La directive *ngClass** (pour l'application d'une class CSS)

```
@Component
({
    selector:'pm-listCourses',
    template:`
    <div *ngFor="let cours of listCours">
        <h1 [ngClass]="'text-success':cours ==='Angular'"> {{cours}}<h1/>
    </div>
    `
})
export class ListCourseComponent{
    listCours=["Cours1","Angular","Cours3"]
}
```

TEMPLATES & DIRECTIVES

Les **directives** structurelle **d'Angular**

- **La directive *ngStyle** (pour l'application d'un Style CSS)

```
@Component
({
    selector:'pm-listCourses',
    template:`
    <div *ngFor="let cours of listCours">
        <h1 [ngStyle]="{'color':cours === 'Angular' ? 'green' : 'red' }"<h1/>
    </div>
    `
})
export class ListCourseComponent{
    listCours=["Cours1","Angular","Cours3"]
}
```

04

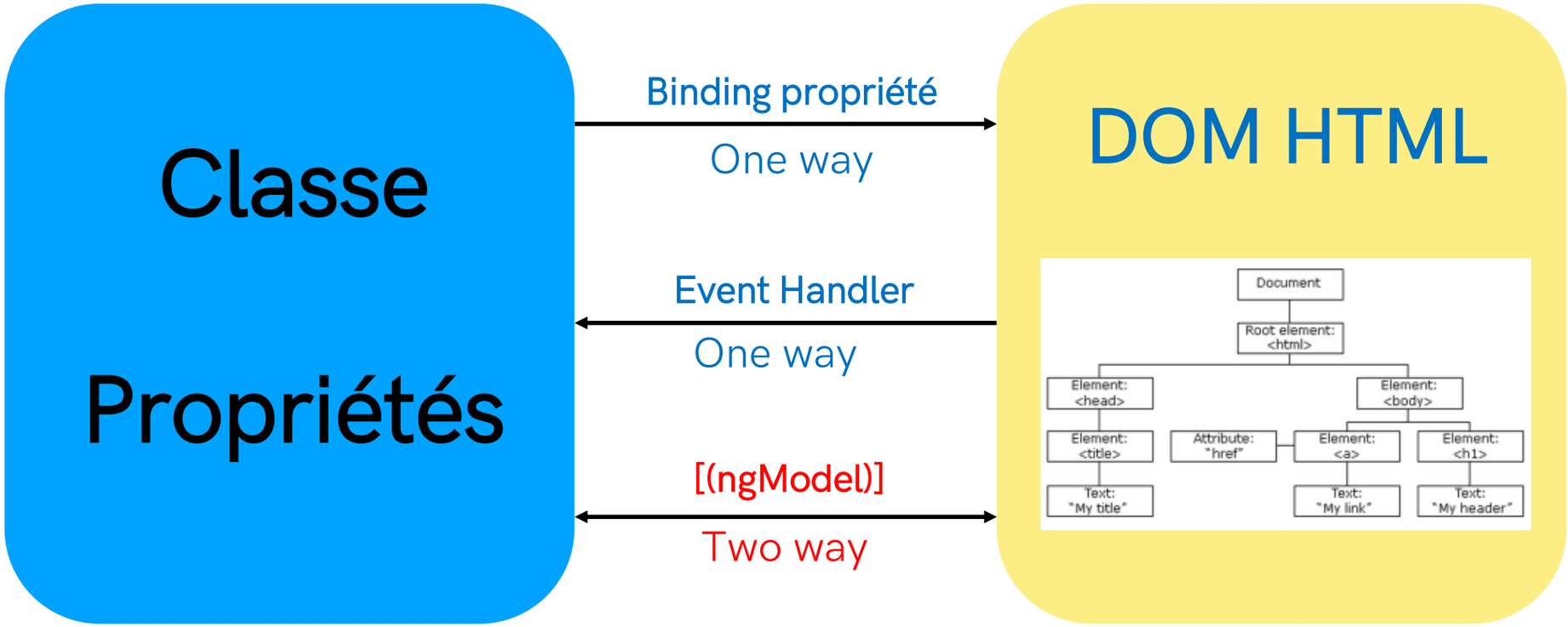
LE BINDING & LES PIPES

Comprendre le fonctionnement du Binding et des Pipes dans Angular

LE BINDING & LES PIPES

Le Property Binding (Liaison de propriété) avec Angular

- Le binding est la communication entre la classe de composant et le template HTML



LE BINDING & LES PIPES

Le Property Binding (Liaison de propriété) avec Angular

- Le binding d'une propriété ou méthode de la classe vers le template HTML se fait avec la syntaxe **{{propriete}}** ou **{{methode()}}** dans le **template HTML**

```
@Component
({
    selector:'pm-listCourses',
    template:`
    <h1> {{titre}} <h1/>
    <h1> {{getDescription()}} <h1/>
    `
})
export class ListCourseComponent{
    titre="La liste des cours";
    getDescription():string
    {
        return "Description des cours"
    }
}
```

LE BINDING & LES PIPES

Les pipes en Angular

- Le principe de pipe est de prendre une donnée en entrée et de lui appliquer un traitement particulier
  - Permet l'automatisation de certains traitement
- Il existe de nombreux Pipes natif dans le framework Angular
  - Uppercase
  - Lowercase
  - Decimal
  - Currency
  - Percent … etc.

LE BINDING & LES PIPES

Les pipes en Angular

- Voici un exemple de traitement avec le Pipe Uppercase
  - ➢ <td>{{ livre.Name | uppercase }}</td>
    - ✓ Uppercase appliquera systématiquement la mise en majuscule de la chaîne livre.Name

Titre
BIG DATA FOR DUMMIES
BIG DATA
DATABASE ENGINEERING

LE BINDING & LES PIPES

Les pipes en Angular

- Voici un exemple de traitement avec le Pipe Currency
  - ➢ <td>{{ livre.Price| currency | uppercase }}</td>
    - ✓ Currency appliquera systématiquement l'unité de prix sur la valeur livre.Price

Prix
$98.00
$120.00

LE BINDING & LES PIPES

Les pipes en Angular

- Voici un exemple de configuration du Pipe Currency
  - ➢ <td>{{ livre.Price| currency : 'EUR' | uppercase }}</td>
    - ✓ Currency appliquera systématiquement l'unité Euro (€) sur la valeur livre.Price

Prix
€98.00
€120.00
€66.00

LE BINDING & LES PIPES

Les pipes en Angular

- Nous pouvons créer nos propres pipes afin d'appliquer notre propre traitement aux données

  - **ng** generate **pipe** <pipeName>

- Une fois créer nous pouvons définir le pipe, ici un pipe qui supprime les espaces vides dans les chaînes de caractères

```
@Pipe({
    name: 'deleteSpace'
})

export class DeleteSpacePipe implements PipeTransform{
    transform(value: string) {
        return value.replace(' ','');
    }
}
```

LE BINDING & LES PIPES

Les pipes en Angular

- Une fois le Pipe défini, nous allons ajouter la classe **DeleteSpacePipe** dans **app.module.ts**

  - Cela permet de le rendre disponible dans toute l'application

```
@NgModule({
  declarations: [
    AppComponent,
    ListCourseComponent,
    DeleteSpacePipe
  ],
```

LE BINDING & LES PIPES

Les pipes en Angular

- Voici un exemple d'utilisation de notre pipe **deleteSpace**
  - ➢ <td>{{ livre.Category | deleteSpace }}</td>
    - ✓ Il supprimera les espaces avant, dans et après la chaîne de caractères

Catégorie
Bigdata
Bigdata

05

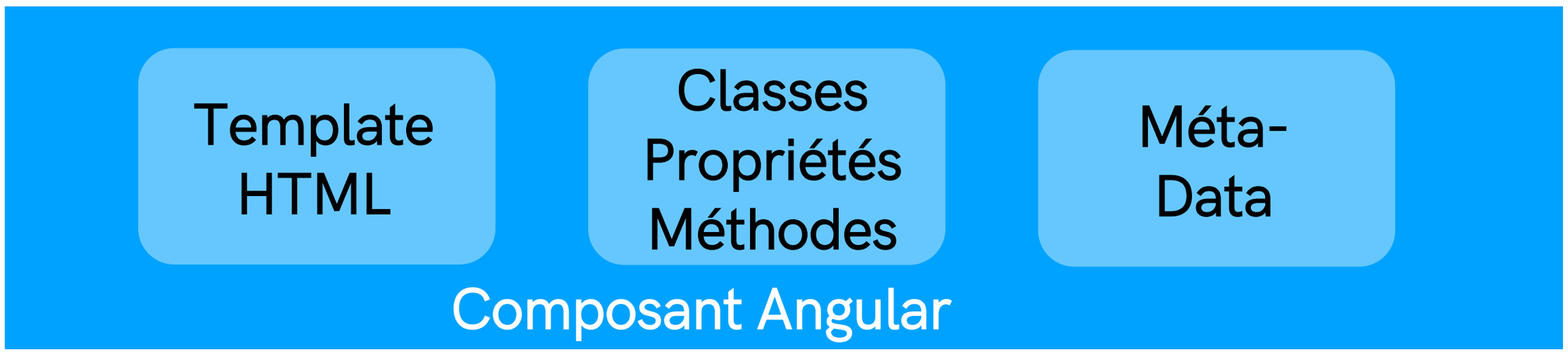
COMPOSANTS & MODULES

Détails des composants et de la communication entre-eux

COMPOSANTS & MODULES

Définition d'un composants en Angular

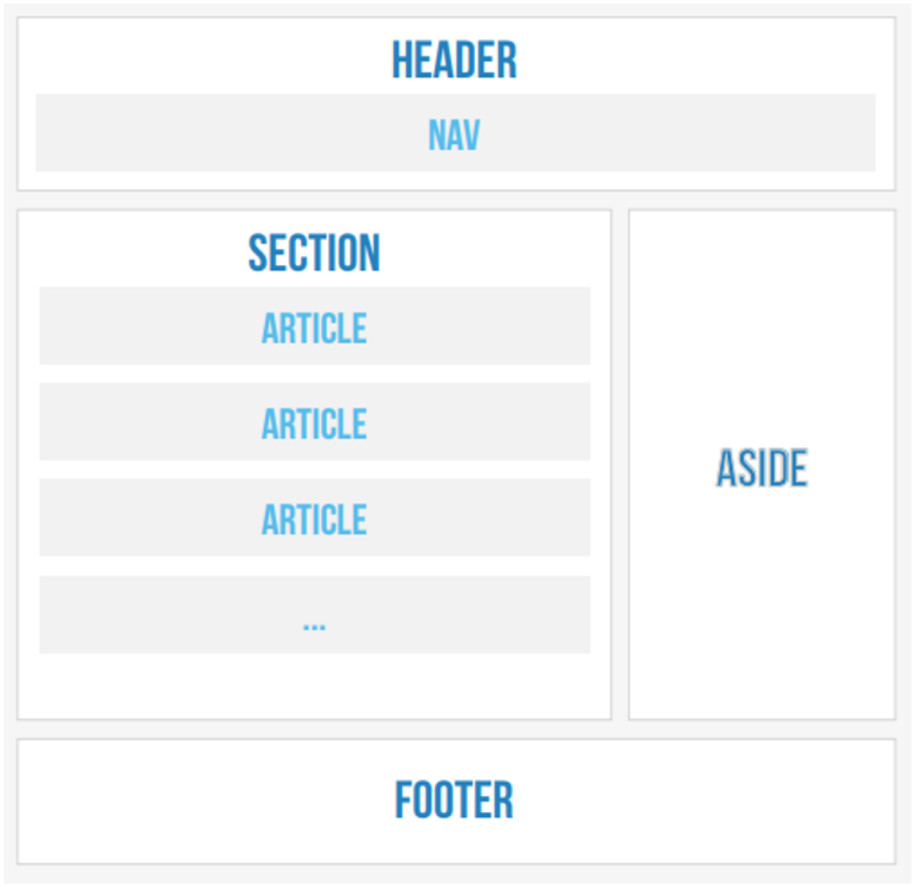
- Comme vu précédemment, pour fonctionner, un composants est constitué de trois éléments (quatre avec le fichier pour les test unitaires). Ces fichiers contiennent :
  - Template HTML
  - Une Classe avec propriétés et méthodes
  - Les Méta-Données



COMPOSANTS & MODULES

Définition d’un composants en Angular

- Les principe d ’une Single Page Application (SPA) et de découper notre page en composants substituables



COMPOSANTS & MODULES

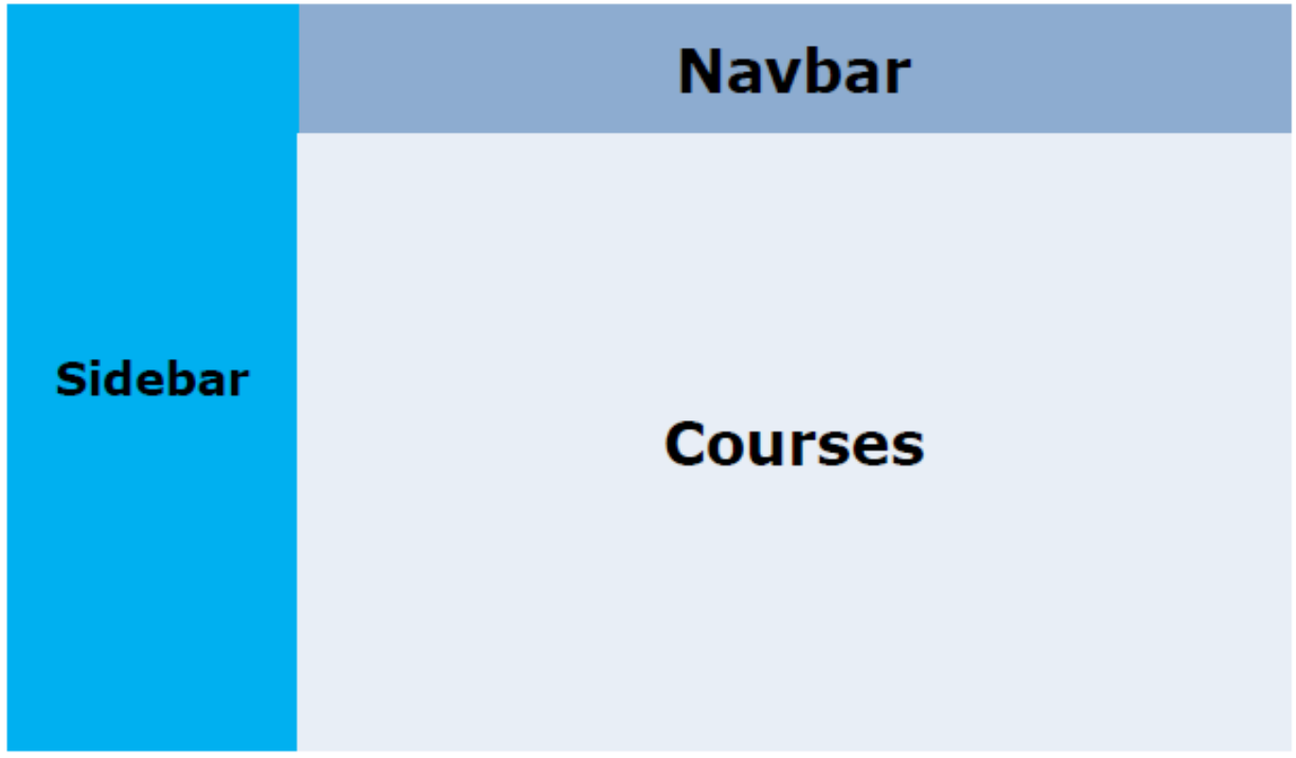
Définition d'une **S**ingle **P**age Application (**SPA**)

- Les avantage d'une SPA sont nombreux
  - Réduire la dépendance avec le serveur
  - Il n'a pas de problème de monter en charge si le nombre d'utilisateurs explose
  - La charge serveur réduite
  - On peut consommer des microservices avec le SPA
  - Réduction des coûts (Développement, maintenant, exploitation)

COMPOSANTS & MODULES

Définition d’un composants en Angular

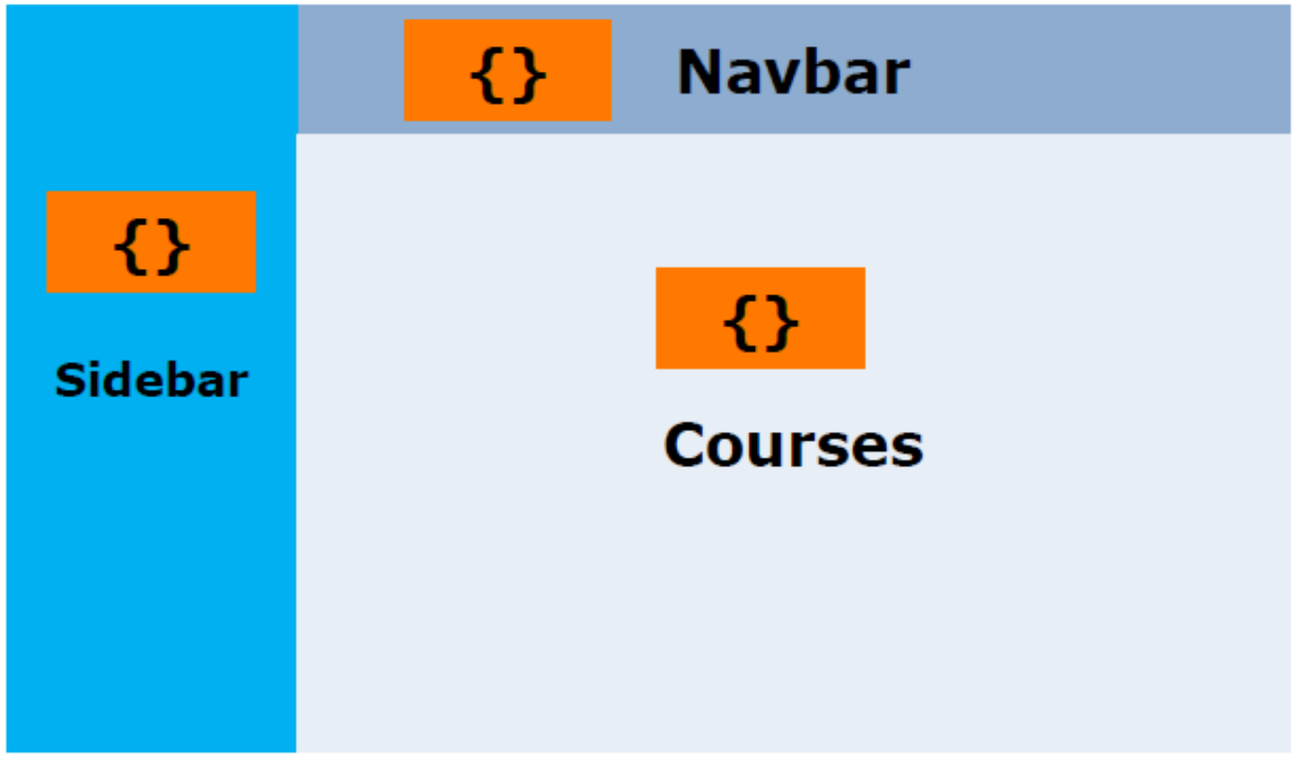
- Dans le cas de notre projet Bibliothèque, on peut envisager de le découper comme suit:



COMPOSANTS & MODULES

Définition d’un composants en Angular

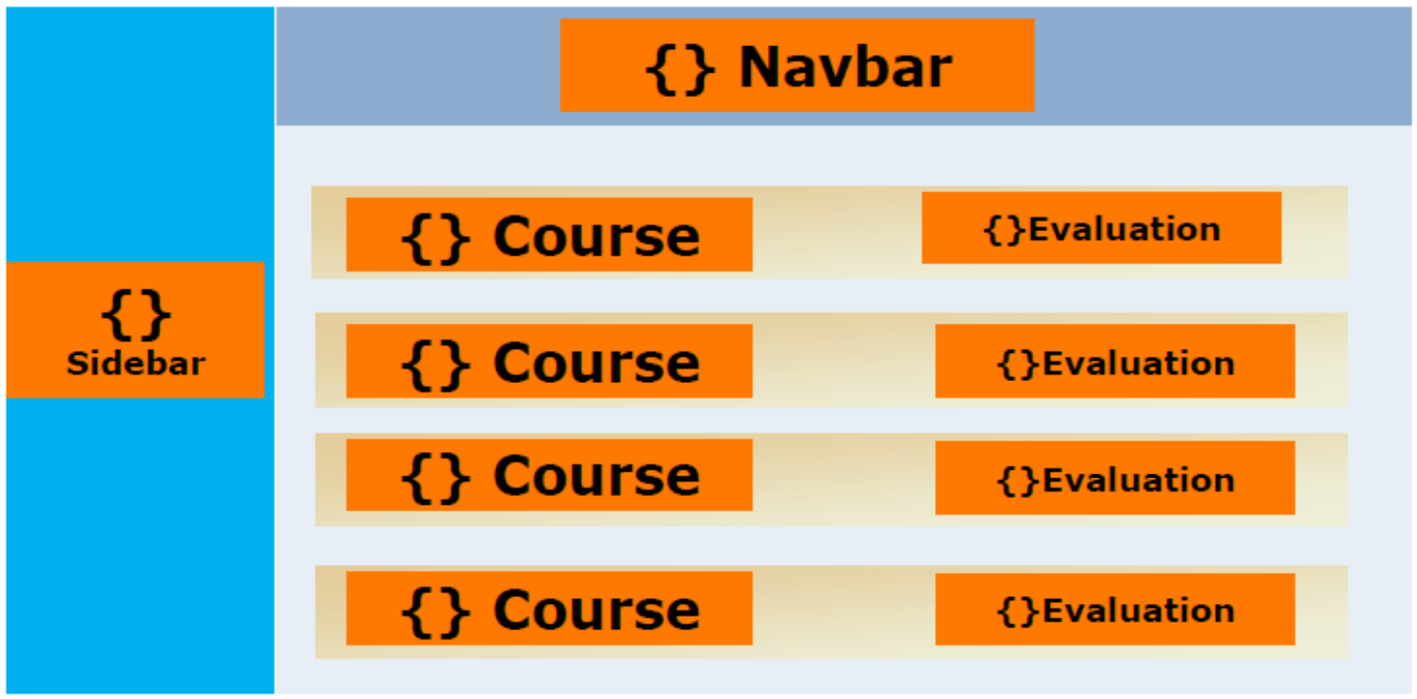
- Mise en place des propriétés à binder:



COMPOSANTS & MODULES

Définition d’un composants en Angular

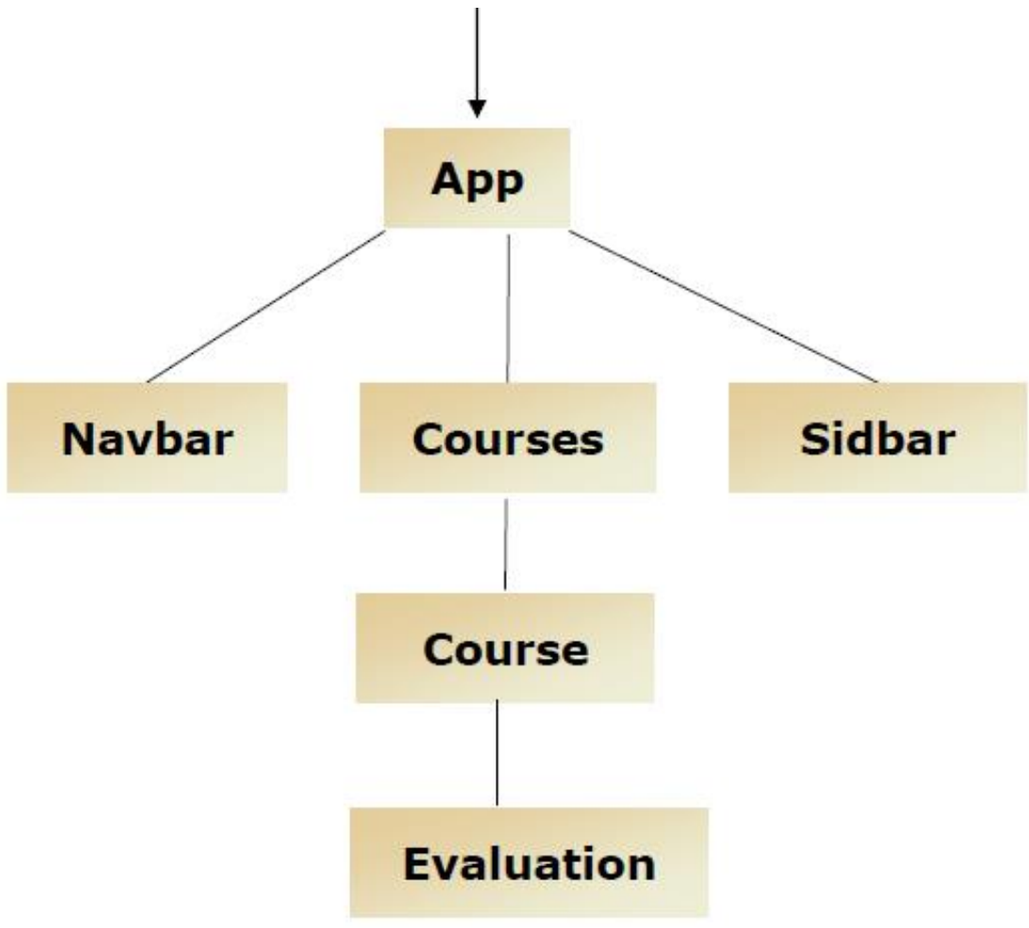
- Création d’un composant Course pour l’affichage de chaque cours (Avec ses propriétés et méthodes).



COMPOSANTS & MODULES

Définition d'un composants en Angular

- Représentation du DOM de notre Application



COMPOSANTS & MODULES

Définition d'un composants en Angular

- Développement du composant Courses (app-cours)

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

@Component
({
    selector:'app_cours',
    template:`
    <h1> {{titre}} </h1>
    `,
    styles:[]
})
export class CourseComponent{
    titre="Angular 11";
}
```

COMPOSANTS & MODULES

Création d'un composants en Angular

- Un composant est défini grâce à l'utilisation du décorateur **@Component**
  - Un décorateur est une fonction qui sera exécutée préalablement à la classe et qui lui fournira du comportement additionnel à celui que l'on a nous même défini.

```
@Component
({
    selector:'app_cours',
    template:`
    <h1> {{titre}} </h1>
    <h2> {{getDescription()}}</h2>
    `,
    styles:[]
})
```

COMPOSANTS & MODULES

Création d'un composants en Angular

- Composant app_cours complété

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

@Component
({
    selector:'app_cours',
    template:`
    <h1> {{titre}} </h1>
    <h2> {{getDescription()}}</h2>
    `,
    styles:[]
})

export class CourseComponent{
    titre="Angular 11"; // propriété
    getDescription()    // Méthode
    {
        return "Les bases indispensable de l'Angular"
    }
}
```

COMPOSANTS & MODULES

Création d'un composants avec **Angular-CLI**

- En utilisant la commande
  - **ng** generate component <nomComposant> (**ng** g c <nom>)
- La CLI nous informe de la création de 4 fichiers et de l'update de app.modules.ts

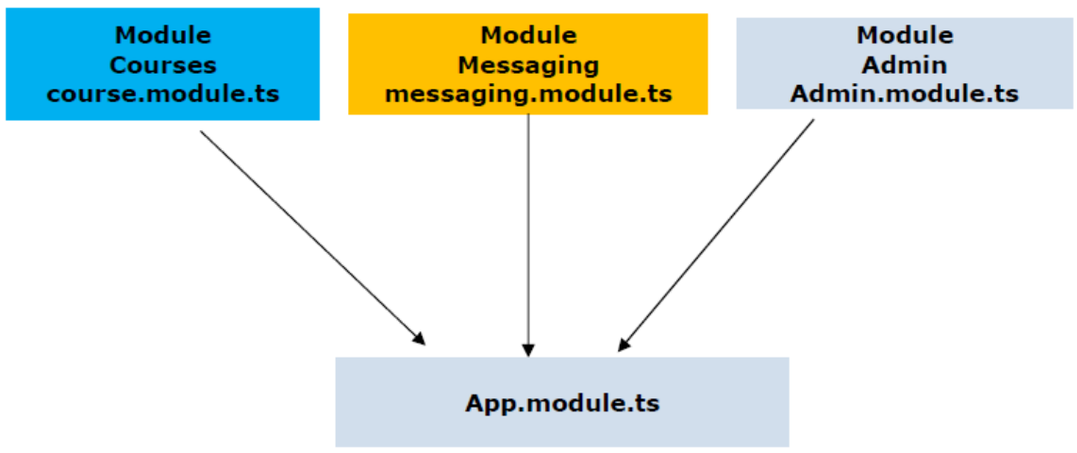
```
$ ng g c MonComposant
CREATE src/app/mon-composant/mon-composant.component.html (28 bytes)
CREATE src/app/mon-composant/mon-composant.component.spec.ts (669 bytes)
CREATE src/app/mon-composant/mon-composant.component.ts (302 bytes)
CREATE src/app/mon-composant/mon-composant.component.css (0 bytes)
UPDATE src/app/app.module.ts (501 bytes)
```

- Le composant créé apparait dans notre arborescence

COMPOSANTS & MODULES

Création d'un **module** de composants

- Les modules permettent de regrouper des composants par thématique
  - Permet une meilleur visibilité des projets volumineux
- Il est possible de ne charger les composants de la page « à la demande » de l'utilisateur
  - Par la mise en place du « Lazy-loading »



COMPOSANTS & MODULES

Création d'un **module** de composants avec **Angular-CLI**

- En utilisant la commande
  - **ng** generate module <nomModule> (**ng** g m <nom>)

- La CLI nous informe de la création du module

```
$ ng generate module MonModule
CREATE src/app/mon-module/mon-module.module.ts (195 bytes)
```

- Le composant créé apparait dans notre arborescence

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';



@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class MonModuleModule { }
```

06

ROUTAGE & NAVIGATION

Mise en place du router et des éléments de navigation Angular

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Une application Angular est une application SPA (Single Page Application), donc il y a une seule page .HTML

- Les différents Template HTML des composants se trouvent dans la page **index.html**

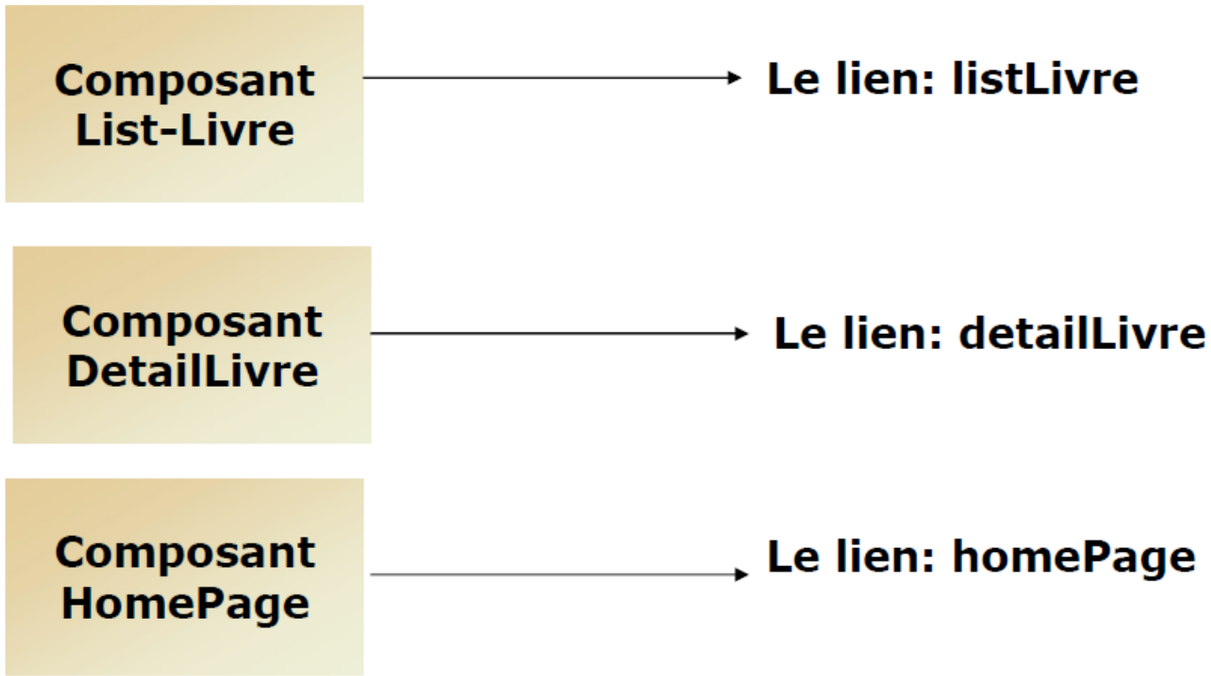
- Nous avons besoin un système de routage pour spécifier chaque composant avec un lien

  - Le **routage** est assuré par la classe **Router** de Angular

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Exemple de besoin de routage pour des composants



ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Pour la mise en place du routage, il faut définir un tableau des routes
  - ➢ Permet de spécifier chaque lien avec son composant

Composant	Link
List-Livre	listLivre
Detail-Livre	detailLivre
Home	homePage

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Dans notre fichier **app.module.ts** il faut procéder à l'importation du **RouterModule**

  - Nous pouvons maintenant créer notre tableaux de constantes **routes**

```
const routes: Routes = [
  ];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Une fois notre tableau de **routes** créé, nous pouvons mettre en place l'ensemble des routes des notre application

  - Chaque **path** correspond à un point d'entrée par l'url du navigateur et redirige vers le component cible

```
const routes: Routes = [
  {path:'',component:HomeComponent},
  {path:'home',component:HomeComponent},
  {path:'livres',component:ListCourseComponent},
  {path:'livre/:id',component:DetailLivreComponent},
  {path:'**',component:PageNotFoundComponent}

  ];
```

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Maintenant que les routes sont fonctionnelles nous pouvons les utiliser dans nos composants

  - Création d'une barre de navigation dans **app.component.html**

```
<nav class="nav navbar-expand navbar-light bg-light">
    <ul class="nav nav-pills">
    <li>
        <a class="nav-link" [routerLink]="['/home']">Accueil</a>

    </li>
    <li>
        <a class="nav-link" [routerLink]="['/livres']">Livres</a>
    </li>
</ul>
</nav>
```

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Les rendu de la cible sera fait à l'intérieur de la balise
  - <router-outlet></router-outlet>

```
<nav class="nav navbar-expand navbar-light bg-light">
    <ul class="nav nav-pills">
    <li>
        <a class="nav-link" [routerLink]="['/home']">Accueil</a>

    </li>
    <li>
        <a class="nav-link" [routerLink]="['/livres']">Livres</a>
    </li>
</ul>
</nav>
<div class="container">
    <router-outlet></router-outlet>
</div>
```

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Parfois, le passage des paramètres est nécessaire dans le routage
  - ➢ Paramètres passé par l'url

  - ➢ Dans le tableau de routes

```
{path:'livre/:id',component:DetailLivreComponent},
```

  - ➢ Utilisation de ce paramètres dans le composant.html

```
<td><a class="btn btn-primary" [routerLink]="['/livre',livre.BookId]"> </a></td>
```

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- La sécurisation des **routes**
  - Avec l'emploi des Guard de Angular

- Création d'un Guard
  - **ng generate guard** <nomGuard>

ROUTAGE & NAVIGATION

Définition du **routage** d'une **application SPA**

- Exemple de **guard detailLivre** pour **limiter** l'accès à un **rôle**
  - Création du guard (**ng generate guard** detailLivre)

```
@Injectable({
  providedIn: 'root'
})
export class DetailLivreGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    return true;
  }

}
```

- Définition de la route à accès limité

```
{path:'livre/:id',component:DetailLivreComponent,canActivate:[DetailLivreGuard]},
```

07

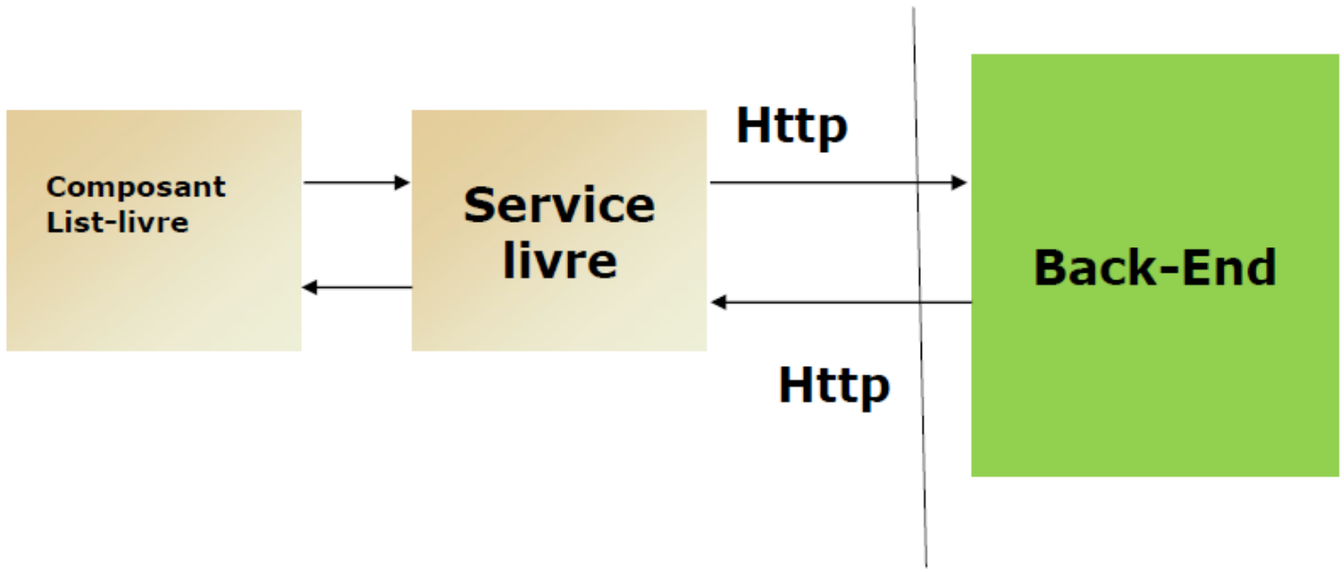
LES SERVICES : HTTP,RXJS

Définition, création et utilisation des bibliothèques RXJS et httpClient

LES SERVICES : HTTP,RXJS

Définition des Services en Angular

- Un service est un intermédiaire entre le back-en et un composant angular
  - ➢ Permet de limiter la solicitation du composant par le back-end
  - ➢ Permet la mise en place de tests unitaires indépendants du back-end



LES SERVICES : HTTP, RXJS

Les **injection** de **dépendances** (**dependency injection**)

- Qu'est-ce qu'une injection de dépendance ?

  - C'est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle.
  - Il consiste à créer dynamiquement les dépendances entre les différents objets

- L'Angular va créer une seule instance de service qui sera partagé dans toute l'application (Singleton = 1 seule instance)

LES SERVICES : HTTP, RXJS

La création d'un **service** dans **Angular**

- Création de la classe
  - Qui utilisera le service
- Création du Service
  - Par l'emploi du décorateur **@Injectable**

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

  constructor() { }
}
```

LES SERVICES : HTTP, RXJS

La création d'un **service** dans **Angular**

- Comment utiliser ce service dans une classe

  - Grace à l'injection de dépendance dans le constructeur de la classe

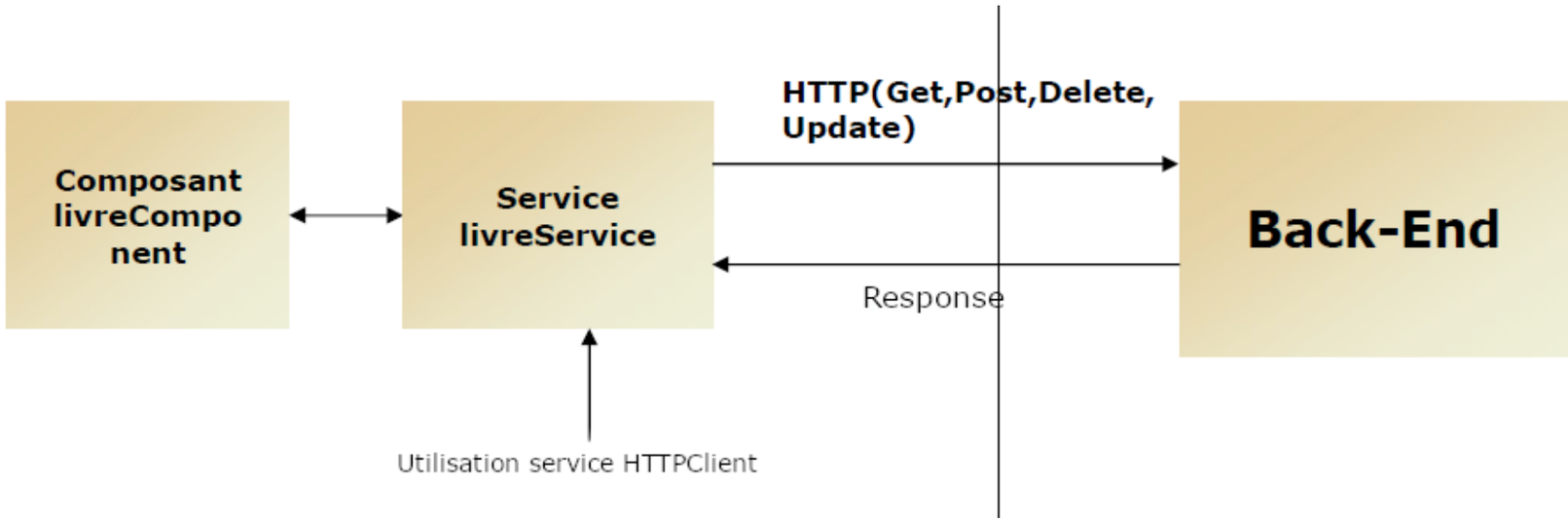
- Exemple de constructeur avec injection de dépendance

```
constructor(private serviceLivre:LivreService) {

}
```

LES SERVICES : HTTP, RXJS

Utilisation d'un **service** dans **Angular**

- Utilisation du service **HTTPCLient** pour utiliser le protocole **HTTP**
  - Permet **d'utiliser** les **4 méthodes** (**Get, Post, Update, Delete**)
- Notre **service livre** utilisera les **Service HTTPClient** pour **communiquer** avec le **Back-end**



LES SERVICES : HTTP, RXJS

Utilisation d'un **service** dans **Angular**

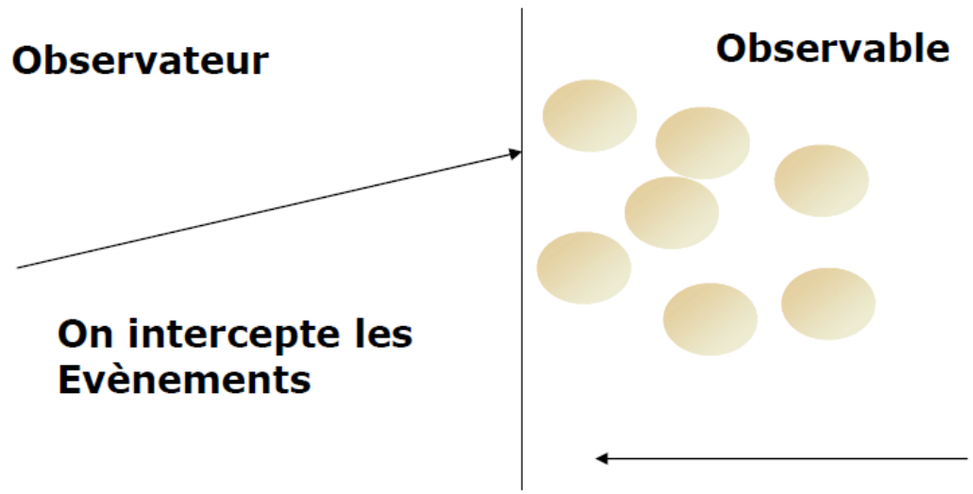
- Exemple pour notre **service Livre**

```
export class LivreService {

  constructor(private http:HttpClient) { }
  getAllLivre():Observable<Ilivre[]>
  {
    return this.http.get<Ilivre[]>('api/livres').pipe
    (
      map(livre=>livre)
    );
  }

}
```

LES SERVICES : HTTP, RXJS

Utilisation d'un **service** dans **Angular RXJS**

- La programmation réactive se base sur le concept d'observateur
  - On définit des observables et des observateurs
- Les observables vont émettre des événements qui seront interceptés par les observateurs

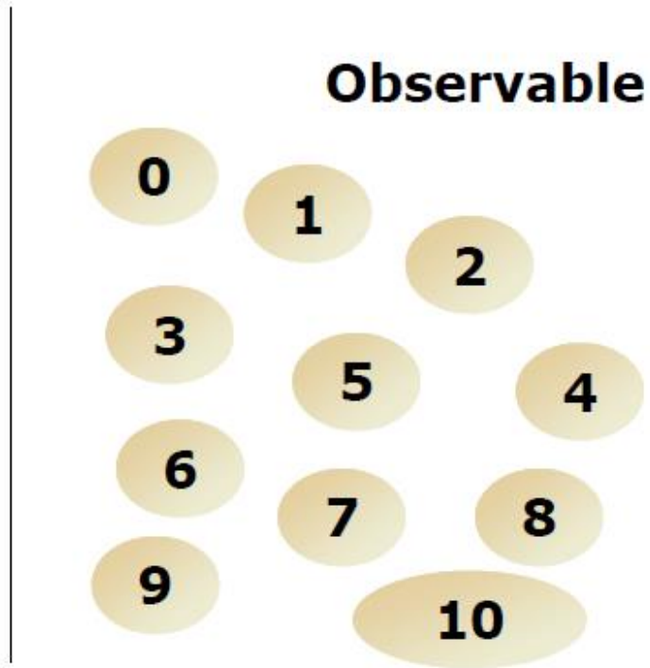


LES SERVICES : HTTP, RXJS

Utilisation d'un **service** dans **Angular RXJS**

- Exemple d'observables

```
source$:Observable<number>=range(0,10);
```



LES SERVICES : HTTP, RXJS

Utilisation d'un **service** dans **Angular RXJS**

- Exemple d'observateur

```
source$:Observable<number>=range(0,10);
```

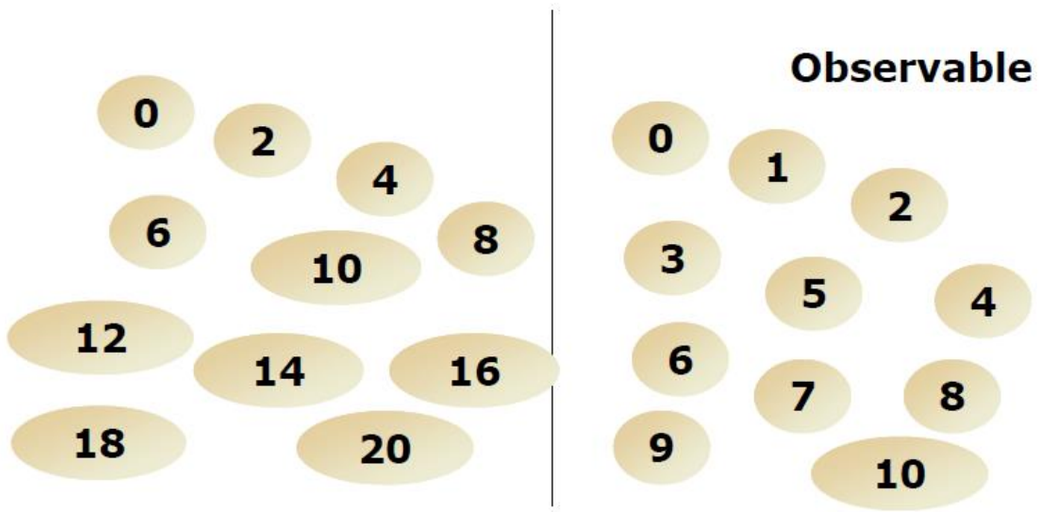
```
this.source$.pipe(
  map(x=>x*2),
).subscribe(
  x=>console.log(x)
)
```

LES SERVICES : HTTP, RXJS

Utilisation d'un **service** dans **Angular RXJS**

- Exemple **fonctionnement** couple **Observateur** / **Observables**

```
this.source$.pipe(
  map(x=>x*2),
).subscribe(
  x=>console.log(x)
)
```

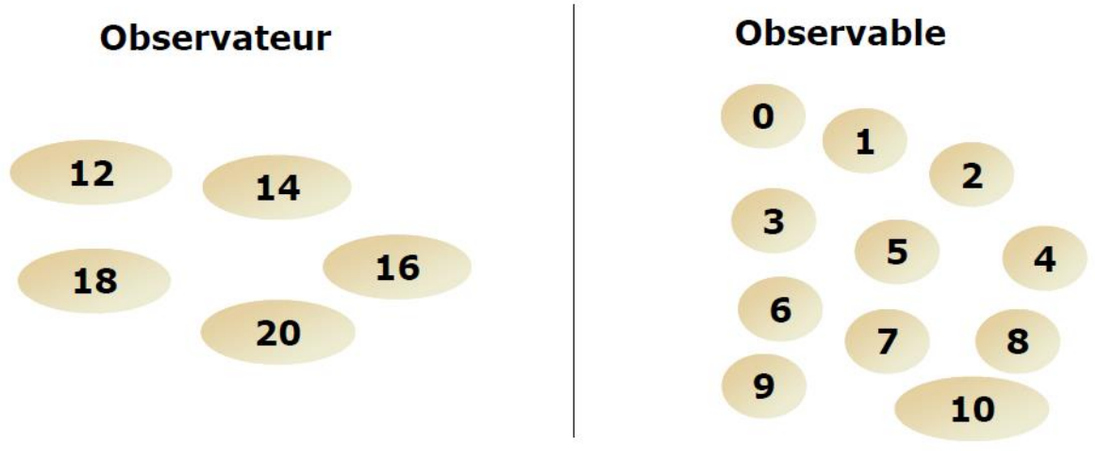


LES SERVICES : HTTP, RXJS

Utilisation d'un **service** dans **Angular RXJS**

- Exemple **fonctionnement** couple **Observateur** / **Observables**

```
this.source$.pipe(
  map(x=>x*2),
  filter(x=>x>10)
).subscribe(
  x=>console.log(x)
)
```



08

LE DEPLOIEMENT

Mise en environnement de production, build et mise en ligne du projet

LE DÉPLOIEMENT EN ANGULAR

- Afin de déployer notre application il nous faut passer par l'étape de Build de notre application

- Webpack nous offre deux possibilité de build

  - Le build avec un environnement de développement
    - ng build
  - Le build avec un environnement de production
    - ng build --prod

- Maintenant notre projet et prêt à être déployé sur un serveur Web.