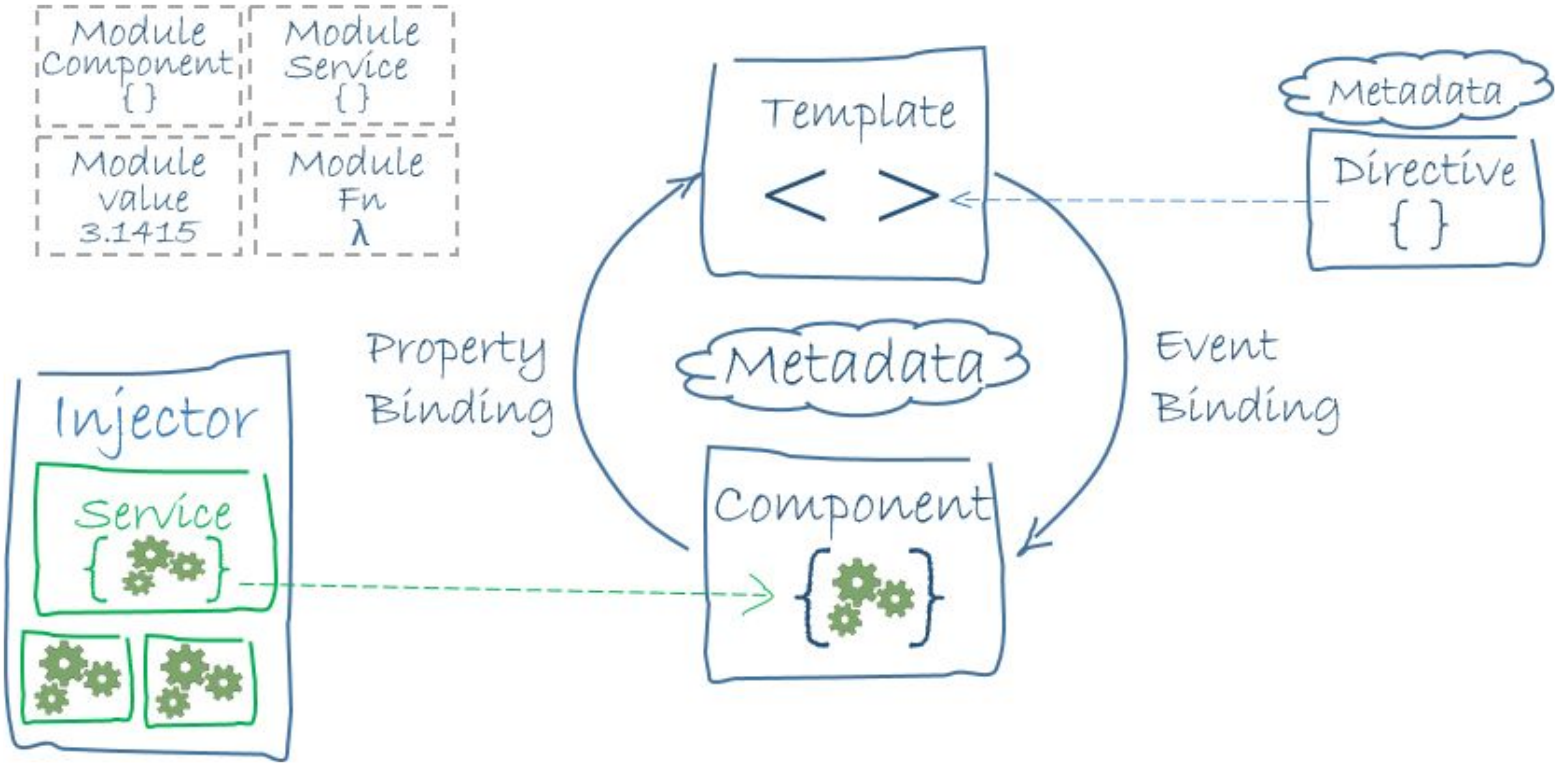

JS_Fighters
Team KLC


A
2.0

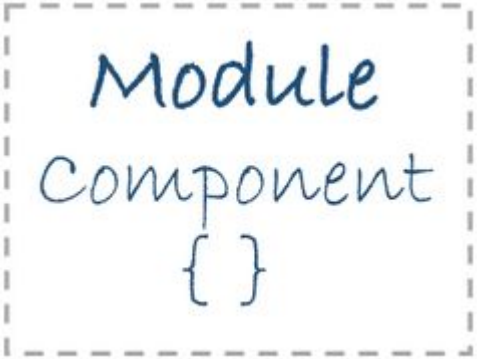

TS

Angular 2 is a framework to help us build client applications in HTML and JavaScript.

The framework consists of several cooperating libraries, some of them core and some optional.

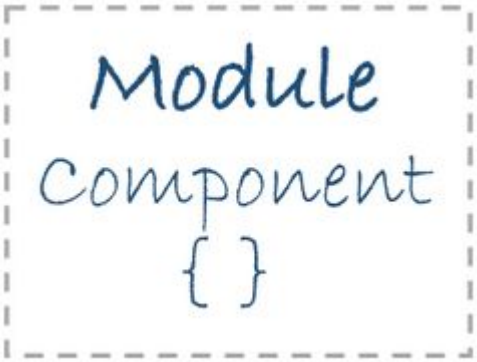

Module
Component
{ }
Module
Service
{ }
Module
value
3.1415
Module
Fn
λ
Template
< >
Metadata
Directive
{ }
Property
Binding
Metadata
Event
Binding
Injector
Service
{ }
Component
{ }

The Module



- Les applications **ANGULAR** sont modulaires (assemblage de plusieur modules).
- Un module est un bloc cohérent dédié à un seul but.
- Un module permet notamment d'exporter des classes.
- Les modules sont optionnels mais recommandé.

The Module



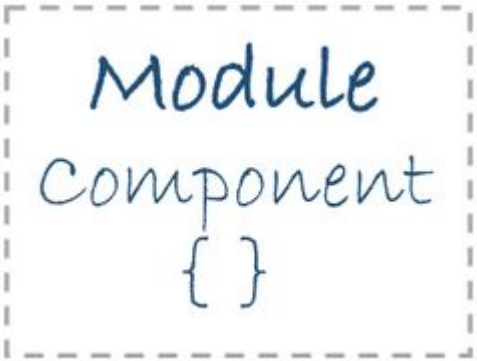
app/app.component.ts (excerpt)

```
export class AppComponent { }
```

app/main.ts (excerpt)

```
import {AppComponent} from './app.component';
```

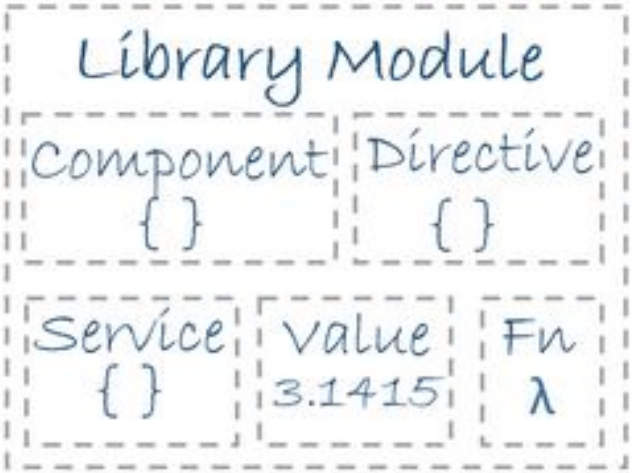
The Module



- **Export** :
  Tells TypeScript that this is a module whose AppComponent class is public and accessible to other modules of the application.

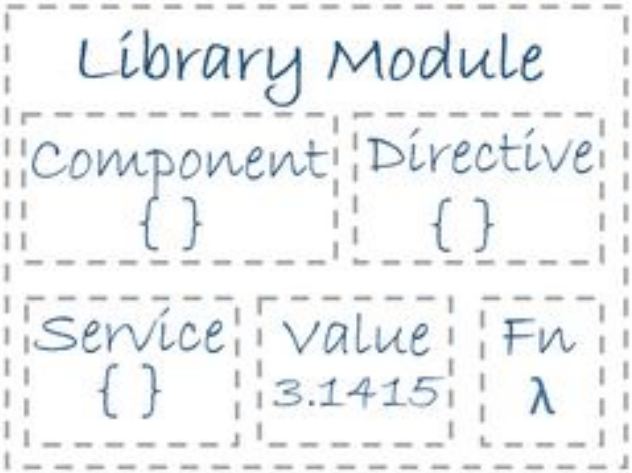
- **Import** :
  Tells the system it can get an AppComponent from a module named app.component located in a neighboring file. The **module name** (AKA module id) is often the same as the filename without its extension.

Library Modules



- Un module peut être composé de plusieur modules. On parle alors de librairie de modules.
- Angular est comme une collection de modules de bibliothèque appelée "**barrels**".
- Chaque bibliothèque est en fait une façade publique sur plusieurs modules privés logiquement liés.

Library Modules

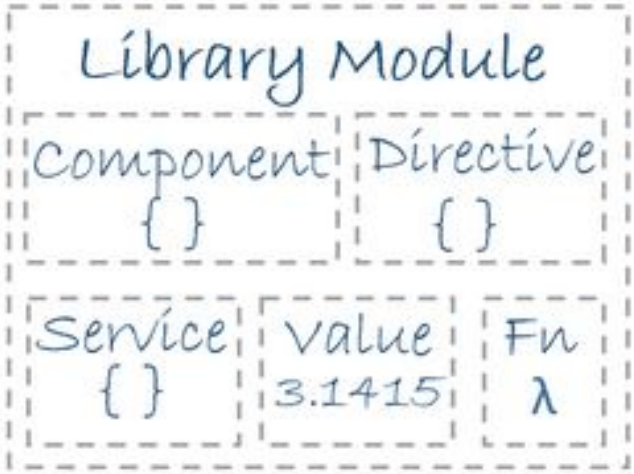



```
import {Component} from 'angular2/core';
```

- Dans le cas présent on import le module **Component** depuis **angular2/core.**

- D'autres librairie existent :
  - **angular2/common**
  - **angular2/router**
  - **angular2/http**
  - **...**

Library Modules




```
import {Component} from 'angular2/core';
```

```
import {AppComponent} from './app.component';
```

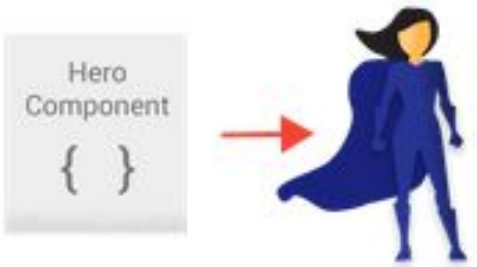
Remarque:

- Les modules d'angular de sont pas préfixé par le chemin.
- Les modules définit doivent être préfixé par le chemin relatif où ils sont enregistrés.

The Component

- Un **Component** controle une **View**.
- La logique de l'application est défini dans un **Component**.
  La classe interagit avec la vue à travers une API de propriétés et de méthodes

The Component



app/hero-list.component.ts

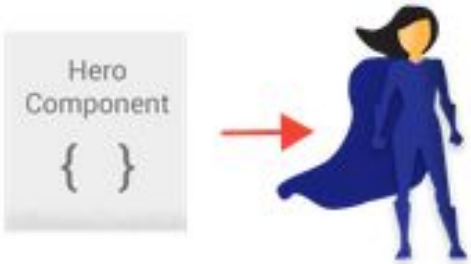
```
1.  export class HeroListComponent implements OnInit {
2.   constructor(private _service: HeroService){ }

3.   heroes:Hero[];
4.   selectedHero: Hero;

5.   ngOnInit(){
6.     this.heroes = this._service.getHeroes();
7.   }

8.   selectHero(hero: Hero) { this.selectedHero = hero; }
9.  }
```

The Component



- Angular cree, met à jours et détruit les **Components** au travers du déplacement de l’ utilisateur dans l’application.

- Le développeur peut agir à chaque instant dans ce cycle de vie par l’option **LifecycleHooks**

The Lifecycle Hooks

- Les instances de **Directive** et de **Component** on un cycle de vie **CRUD**
- Les developpeurs peuvent implémenter ces hooks disponible depuis **angular2/core**
- Aucune **Directive** ou **Component** n' implementera tous ces hooks.
- Certain sont spécifique au **Component**

- **OnInit**
- **OnDestroy**
- **DoCheck**
- **OnChanges**
- **AfterContentInit**
- **AfterContentChecked**
- **AfterViewInit**
- **AfterViewChecked**

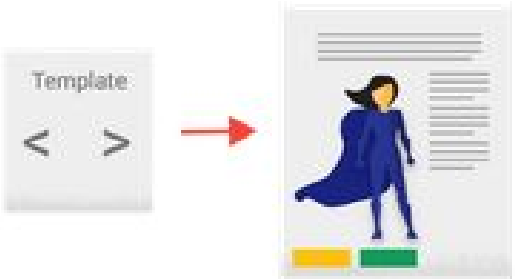
The Lifecycle Hooks

Chaque interface a une unique **hook** méthode. Cette **hook** méthode est nommé par le nom de l'inteface préfixé par **ng**

- **ngOnChanges**
  called when an input or output binding value changes
- **ngOnInit**
  after the first ngOnChanges
- **ngDoCheck**
  developer's custom change detection
- **ngAfterContentInit**
  after component content initialized
- **ngAfterContentChecked**
  after every check of component content
- **ngAfterViewInit**
  after component's view(s) are initialized
- **ngAfterViewChecked**
  after every check of a component's view (s)
- **ngOnDestroy**
  just before the directive is destroyed.

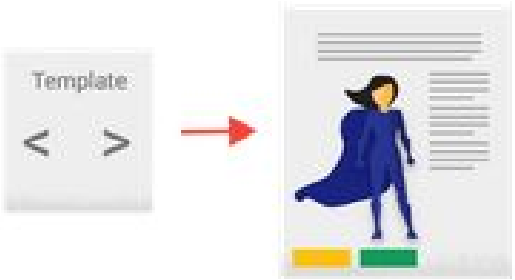
https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html

The Template



- Un **Template** est le companion de la **View** controlé par le **Component.**
- Fichier **HTML**

The Template



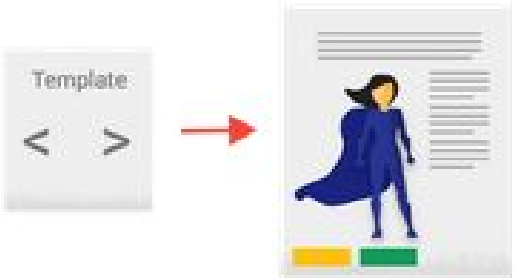
app/hero-list.component.html

```
1. <h2>Hero List</h2>

2. <p><i>Pick a hero from the list</i></p>
3. <div *ngFor="#hero of heroes" (click)="selectHero(hero)">
4.  {{hero.name}}
5. </div>

6. <hero-detail
   *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>
```

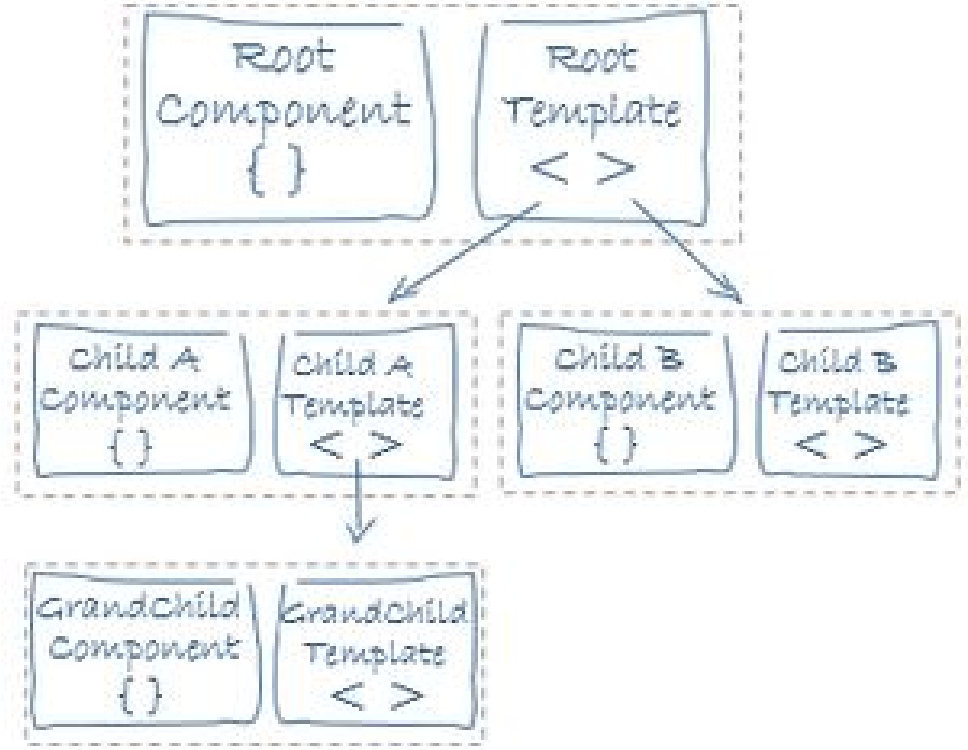
The Template



- **<hero-detail>** est un élément représentant **HeroDetailComponent**
- **HeroDetailComponent** (code non représenté) affiche les infos du héro sélectionné dans **HeroListComponent**
- **HeroDetailComponent** est un fils de **HeroListComponent**

https://angular.io/docs/ts/latest/guide/template-syntax.html

The Template



Angular Metadata

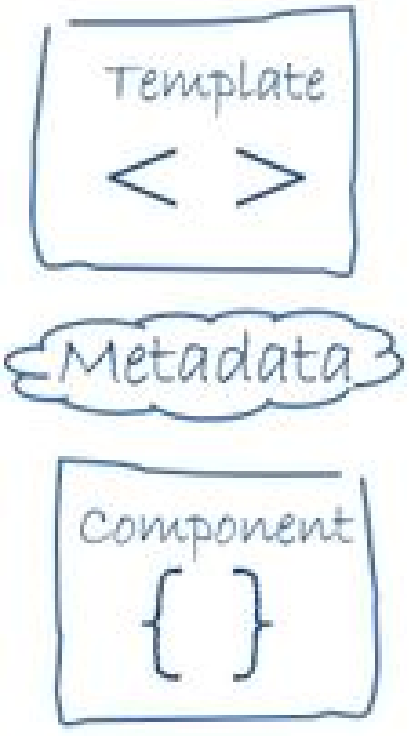
- Les métadonnées indique à angular comment traiter une classe.
- L'attachement des métadonnées en **TypeScript** se fait grâce à un décorateur

Angular Metadata

app/hero-list.component.ts (metadata)

```
1.  @Component({
2.   selector:    'hero-list',
3.   templateUrl: 'app/hero-list.component.html',
4.   directives:  [HeroDetailComponent],
5.   providers:   [HeroService]
6.  })
7.  export class HeroesComponent { ... }
```

Angular Metadata

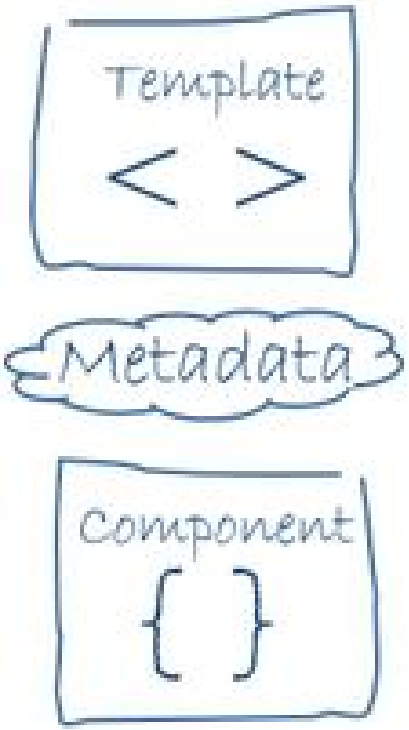


- Le décorateur **@Component** identifie la classe

- **selector**
  Un sélecteur CSS qui indique à Angular de créer et d' inserer l'instance du composant lorsqu'il rencontre la balise **<nameComponent> (<hero-list>)**

- **templateUrl**
  L'adresse du **Component**

Angular Metadata

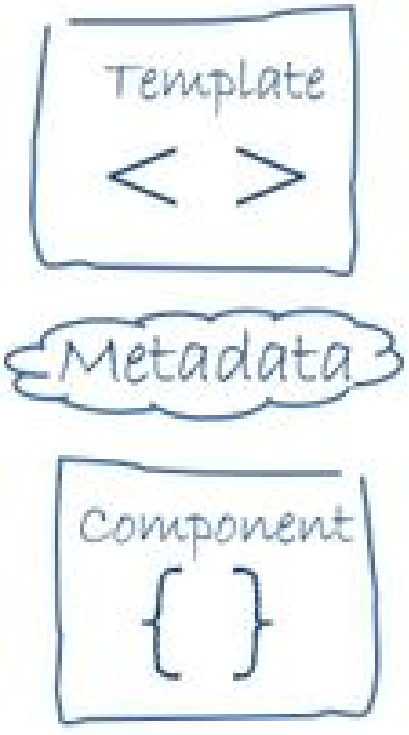


- **directives**
  Un tableau de **Component** ou de **Directive** que requiere le **Template**.

- **providers**
  Un tableau de **dependency injection providers** (fournisseur d'injection de dépendance) pour les services que le **Component** requiere.

- **templateUrl**
  L'adresse du **Component**

Angular Metadata

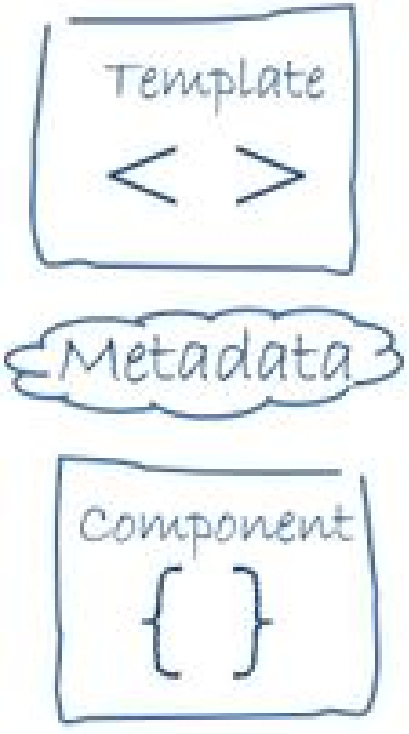


La fonction **@Component** prend l'objet de configuration et le transforme en métadonnées qu'il attache à la définition de la classe du **Component**.

Angular découvre ces métadonnées à l'exécution et sait donc quoi faire.

Template, Metadata, Component ensemble décrive **View**

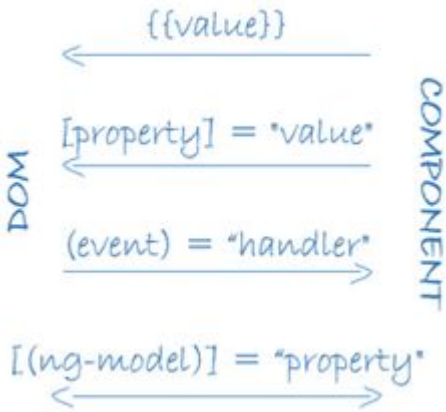
Angular Metadata



D’autre décorateurs de métadonnées existent et s’ implemente de manière similaire :

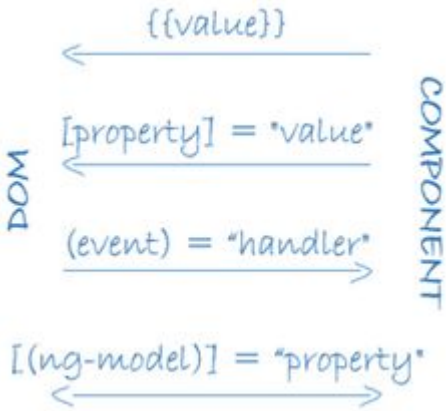
- **@Injectable**
- **@input**
- **@output**
- **@RouterConfig**
- **...**

Data Binding (liaison de données)



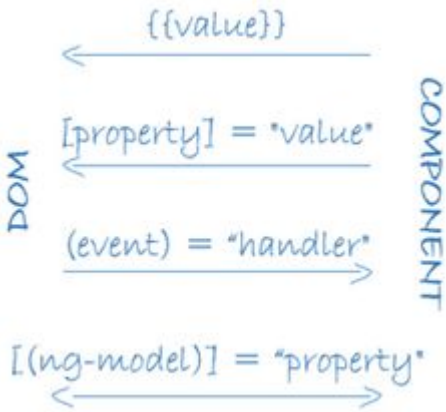
- Sans Framework, nous serions responsables de pousser les valeurs de données dans les contrôles HTML.
- Les réponses utilisateurs seraient traduites par des mises à jours de valeurs ...
- Écrire cette logique à la main est simple mais fastidieuse et sujette aux erreurs

Data Binding (liaison de données)



- Angular support le **Data Binding**
- C'est un mécanisme de coordination des parties d' un **Template** avec une partie des **Component**.
- Le **Data Binding** d'angular se fait dans les deux sens grâce notamment à la génaration des balises **Component**.

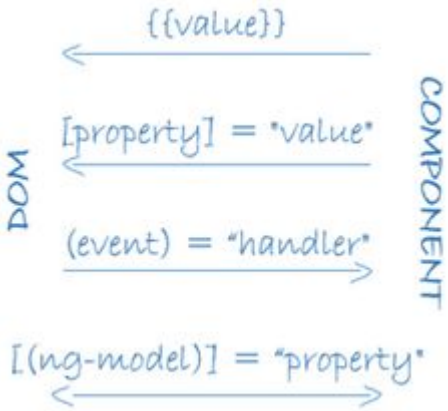
Data Binding (liaison de données)



- Il existe quatre formes de Data Binding
  - interpolation
  - property binding
  - event binding
  - two-way data binding

- Chaque forme à une direction
  - vers le DOM
  - depuis le DOM
  - dans les deux sens

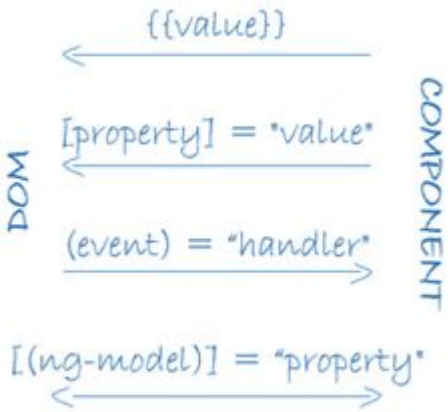
Data Binding (liaison de données)



app/hero-list.component.html (excerpt)

```
<div>{{hero.name}}</div>
<hero-detail [hero]="selectedHero"></hero-detail>
<div (click)="selectHero(hero)"></div>
```

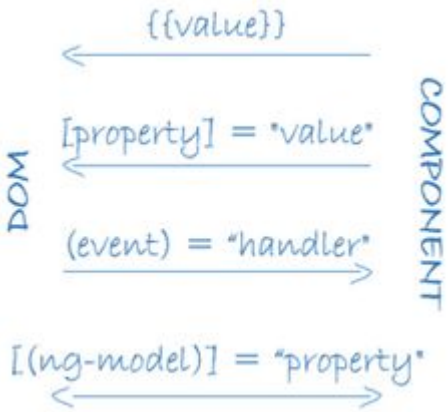
Data Binding (liaison de données)



- **Interpolation**
  Affiche la valeur de la propriété hero.name du composant dans la balise

- **Property binding**
  [hero] envoie selectedHero du parent HeroListComponent à la propriété du fils HeroDetailComponent

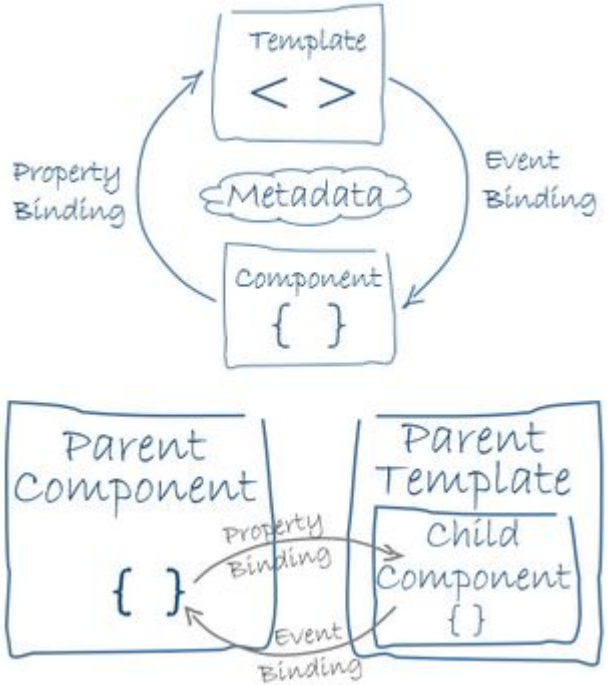
Data Binding (liaison de données)



- **Event binding**
  La méthode selectHero est appelé lorsque l'utilisateur clique sur le nom d'un héro

- **Two-way data binding**
  Combine property et event binding en une unique notation utilisant la directive **ngModel**.

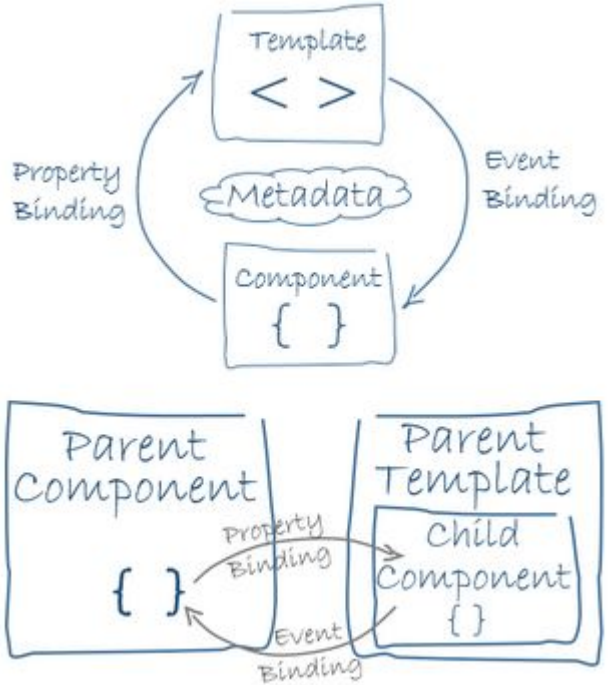
Two-Way Binding (liaison bidirectionnel)




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

- La valeur de propriété d'un objet est envoyé dans la boîte d'entrée du **Component** (property binding).
- Les modifications de l'utilisateur envoie un retour au **Component** qui met à jour la propriété (event binding).

Two-Way Binding (liaison bidirectionnel)



Angular traite toutes les liaisons de données une fois par cycle d'événements JavaScript, des feuilles vers la racine de l'arbre de **Component** de l’application.

The Directive

- Les **Templates** Angular sont dynamique
- Lors que Angular les rends, il transforme le DOM selon les instructions données par une **Directive**
- Une **Directive** est une classe avec des métadonnées de **Directive**.
- Dans **TypeScript** nous applicons le décorateur **@Directive** pour attacher les métadonnées à la classe

The Directive

- Un **Component** est une forme de **Directive**
- Un **Component** est une **Directive** avec un **Template**
- Le décorateur **@Component** est un décourateur **@Directive** étendu avec des fonctionnalités **template-oriented**

The Directive

- Il existe deux autres types de directives
  - Structural
  - Attribute

- Elles apparaissent dans une balise d'élément comme attributs, parfois par leurs noms, mais plus souvent comme l’objet d’une affection ou d'une liaison.

The Directive Structural

Modifient la mise en page en ajoutant, supprimant, et le remplacement des éléments dans le DOM

The Directive Structural


```
<div *ngFor="#hero of heroes"></div>
<hero-detail *ngIf="selectedHero"></hero-detail>
```

- * NgFor
  Ecrase par hero dans la liste heroes
- * NgIf
  Inclut le **Component** HeroDetail uniquement si selectedHero existe.

The Directive Attribute

Modifient l'apparence ou le comportement d'un élément existant.

Dans les **Templates** ils ressemblent à des attributs HTML réguliers, d'où le nom.

The Directive Attribute


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

Il modifie le comportement d'un élément existant (généralement un <input>) en définissant sa propriété de valeur d'affichage et de répondre au changement des événements.

The Directive

- Angular implemente d’autres directives qui soit modifient la structure de mise en page (par exemple ngSwitch) ou modifient les aspects des éléments et des composants DOM (par exemple ng Style et ng classe).

- Et bien sûr, nous pouvons écrire nos propres directives.

The Service

- **Service** est une vaste catégorie qui englobe toute valeur, fonction ou caractéristique des besoins de l'application.
- Presque tout peut être un service.
- Un service est généralement une classe avec un but cadré et bien défini.

The Service

Examples include:

- logging service
- data service
- message bus
- tax calculator
- application configuration

The Service

- Il n'y a rien de spécifique dans Angular 2.0 sur les services.
- Angular 2.0 n'a pas de définition d'un service. Il n'y a pas de classe ServiceBase.
- Pourtant, les services sont essentiels à toute application Angular.

The Service

app/logger.service.ts (class only)

```
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

The Service

app/hero.service.ts (class only)

```
export class HeroService {
  constructor(
    private _backend: BackendService,
    private _logger: Logger) { }

  private _heroes:Hero[] = [];

  getHeroes() {
    this._backend.getAll(Hero).then( (heroes:Hero[]) => {
      this._logger.log(`Fetched ${heroes.length} heroes.`);
      this._heroes.push(...heroes); // fill cache
    });
    return this._heroes;
  }
}
```

The Service

- Voici un HeroService qui récupère les héros et les renvoie dans une promesse résolue.
- Le Service Hero dépend de la LoggerService et un autre BackendService qui gère le travail serveur communication grognement.

The Service

- Les services sont partout. Nos **Components** sont de grands consommateurs de services.
- Ils dépendent de services pour gérer la plupart des tâches.
- Ils ne vont chercher les données du serveur, ils ne valident pas l'entrée d'utilisateur, ils ne se connectent pas directement à la console.
- Ils délèguent ces tâches à des services.

Dependency Injection

- L’injection de dépendance est un moyen de fournir une nouvelle instance d'une classe avec les dépendances entièrement formées dont elle a besoin.

- La plupart des dépendances sont des services.

Dependency Injection

- Angular utilise **l'injection de dépendance** pour fournir de nouveaux **components** avec les services dont ils ont besoin.

- Dans TypeScript, Angular peut dire quels services a besoin de composants en regardant les types de ses paramètres de constructeur.

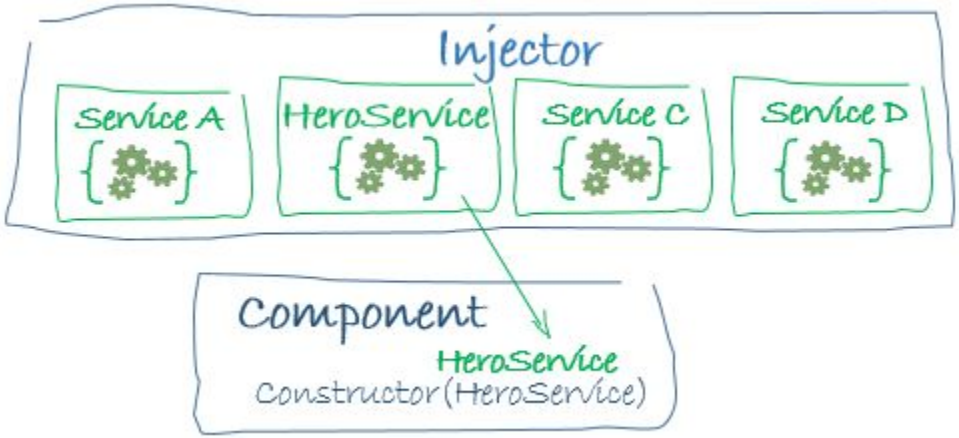
Dependency Injection

app/hero-list.component (constructor)

```
constructor(private _service: HeroService){ }
```

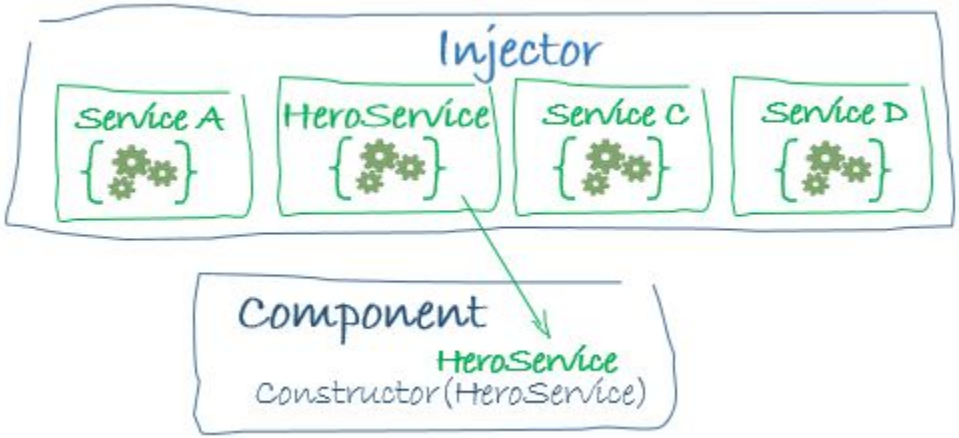
Le constructeur a besoin du HeroService

Dependency Injection



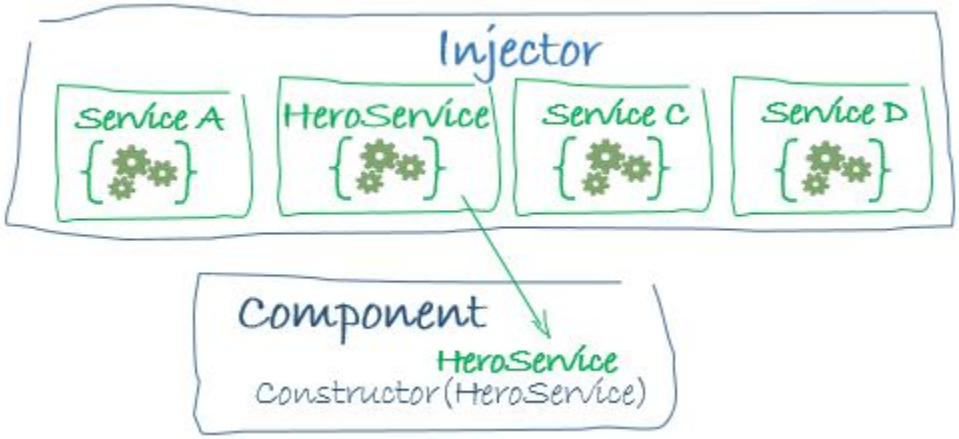
- Lorsque Angular crée un composant, il demande d'abord un **Injector** pour les **services** que le **component** a besoin.
- Un Injector maintient un conteneur des instances de service qu'il a créés précédemment.
- Si une instance de service demandée se trouve pas dans le récipient, l'injecteur instancie une et l'ajoute au récipient avant de renvoyer le service

Dependency Injection



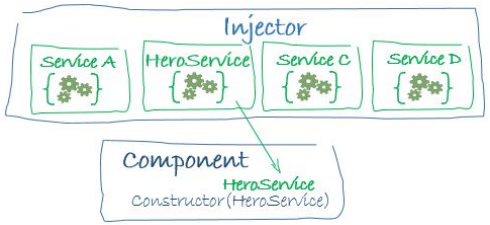
- Lorsque tous les services demandés ont été résolus et retourné, Angular peut appeler le constructeur du composant avec ces services comme arguments.

Dependency Injection



- Un **provider** est quelque chose qui peut créer ou retourner un service, généralement la classe de service lui-même.
- Nous pouvons enregistrer les **providers** à tout niveau de l'arborescence des composants d'application.
- Nous faisons souvent si à la racine de sorte que la même instance d'un service est disponible partout.

Dependency Injection



app/main.ts (excerpt)

```
bootstrap(AppComponent, [BackendService,
HeroService, Logger]);
```

app/hero-list.component.ts (excerpt)

```
@Component({
  providers:   [HeroService]
})
export class HeroesComponent { ... }
//Alternatively, we might register at a component level
```