

# Angular



# Definizione

---

- Angular è un framework, open-source per lo sviluppo di Single Page Application
- Angular è una completa riscrittura, in TypeScript, di AngularJS, da parte dello stesso team di sviluppo
- È un progetto mantenuto principalmente da Google
- Fa parte dello stack MEAN (MongoDB, Express, Angular, NodeJS)

# Linguaggio

---

- È possibile sviluppare applicazioni in:
  - TypeScript
  - Dart
  - JavaScript
- Sebbene venga lasciata libertà di scelta agli sviluppatori, è consigliabile l'utilizzo di TypeScript

# Angular CLI

---

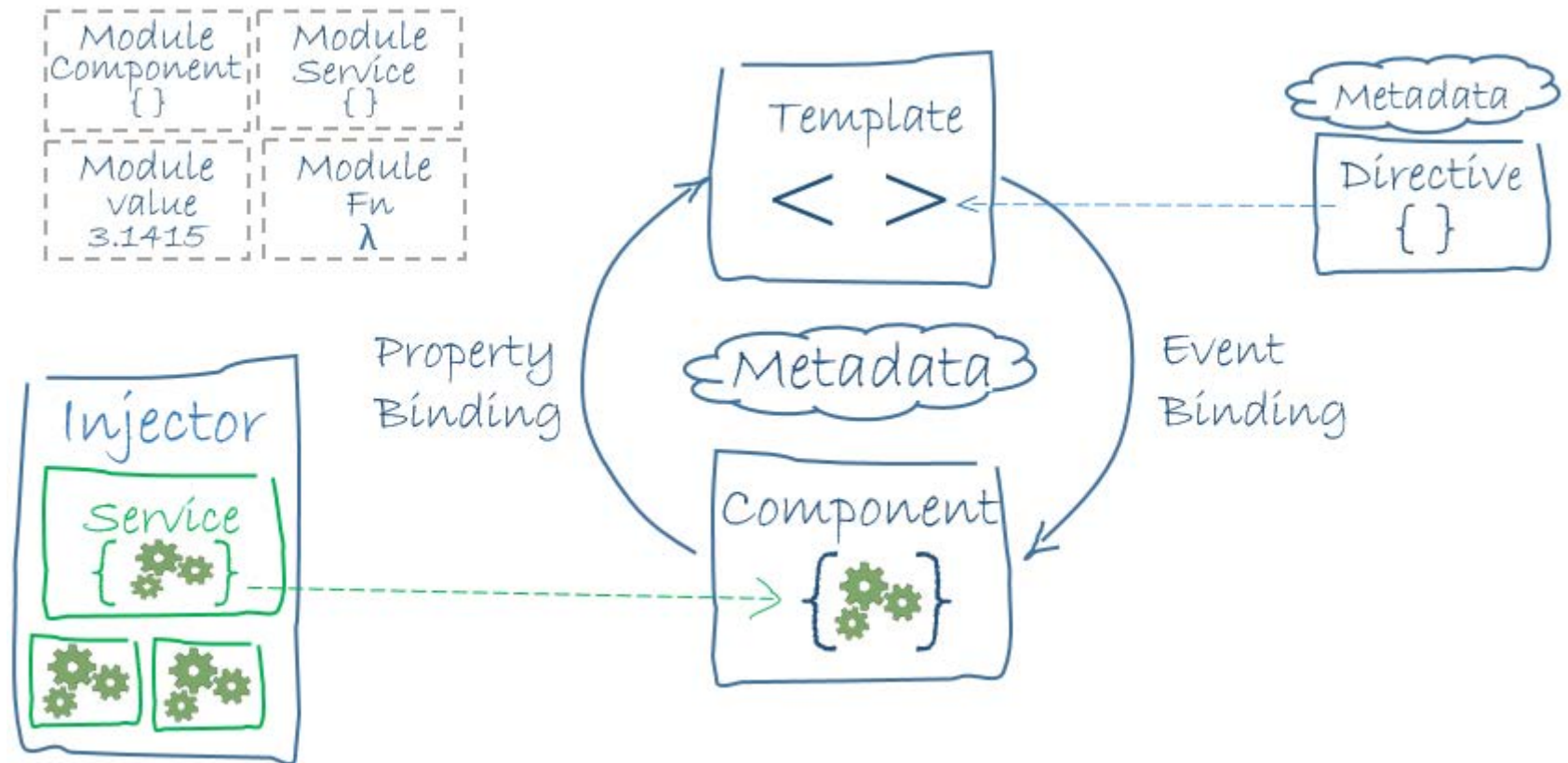
- È disponibile una ***Command Line Interface***, che semplifica la creazione della struttura dell'applicazione
- È basata su **Node**
- È installabile eseguendo il comando:  
`npm install -g angular-cli`

# Angular CLI: Comandi principali

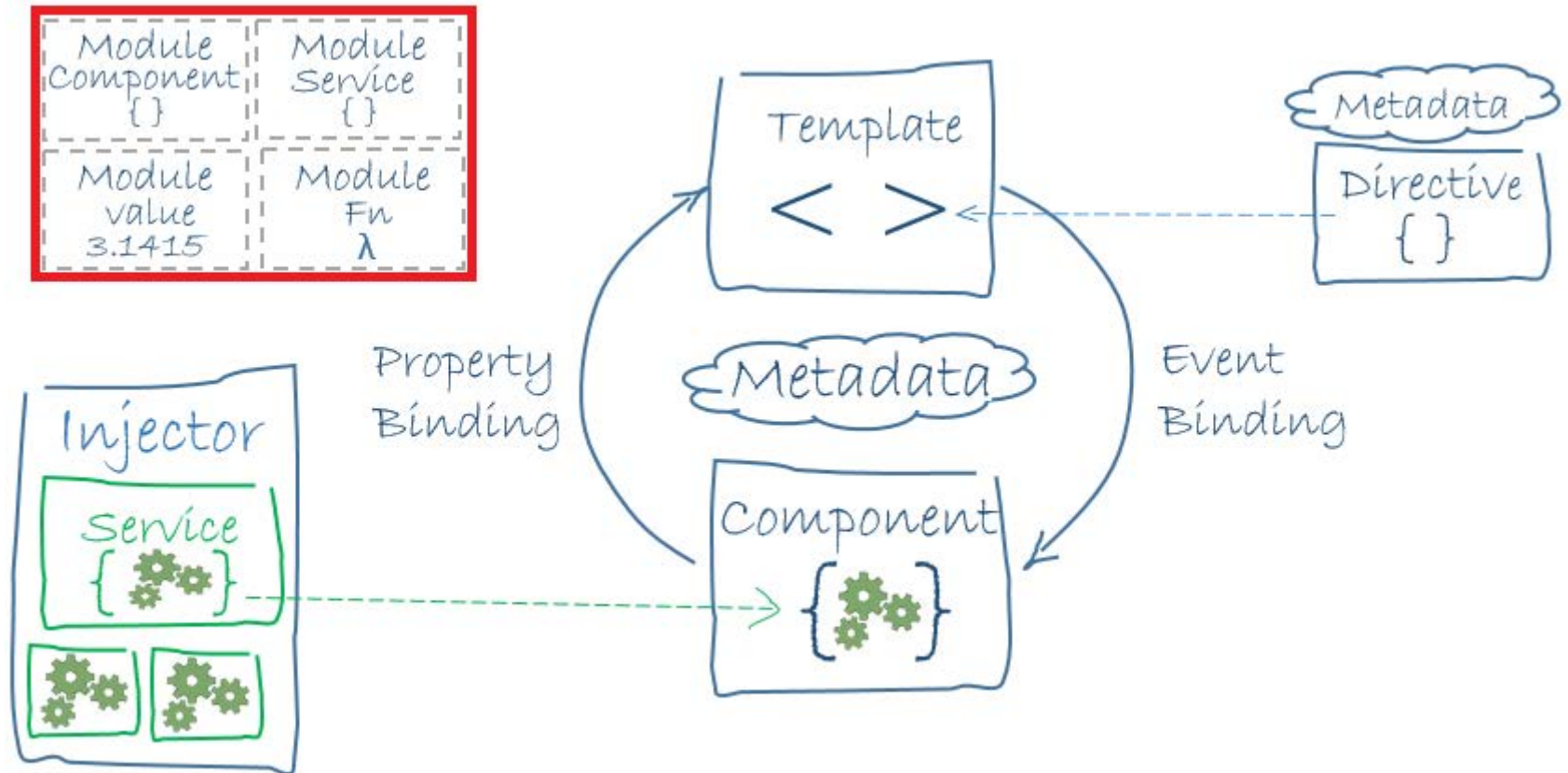
---

- **ng new appname**: creazione di un nuovo progetto
- **ng serve --open**: servire l'applicazione (NB: è necessario entrare nella cartella del progetto: **cd appname**)
- **ng build --prod --bh /myUrl/**: serve per eseguire il build dell'applicazione

# Architettura



# Modules



# Modules

---

- Angular ha un proprio sistema di modularità, chiamato ***Angular modules*** o ***NgModules***
- Ogni applicazione ha almeno un modulo Angular, il modulo ***root***, solitamente chiamato ***AppModule***
- Un NgModule è una classe con un decoratore **@NgModule**, che prende in input un oggetto **metadata**, le cui proprietà descrivono il modulo



# Modules - Esempio

---

```
import { HttpClientModule } from '@angular/http';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    HttpClientModule,
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

# Principali proprietà @NgModule

---

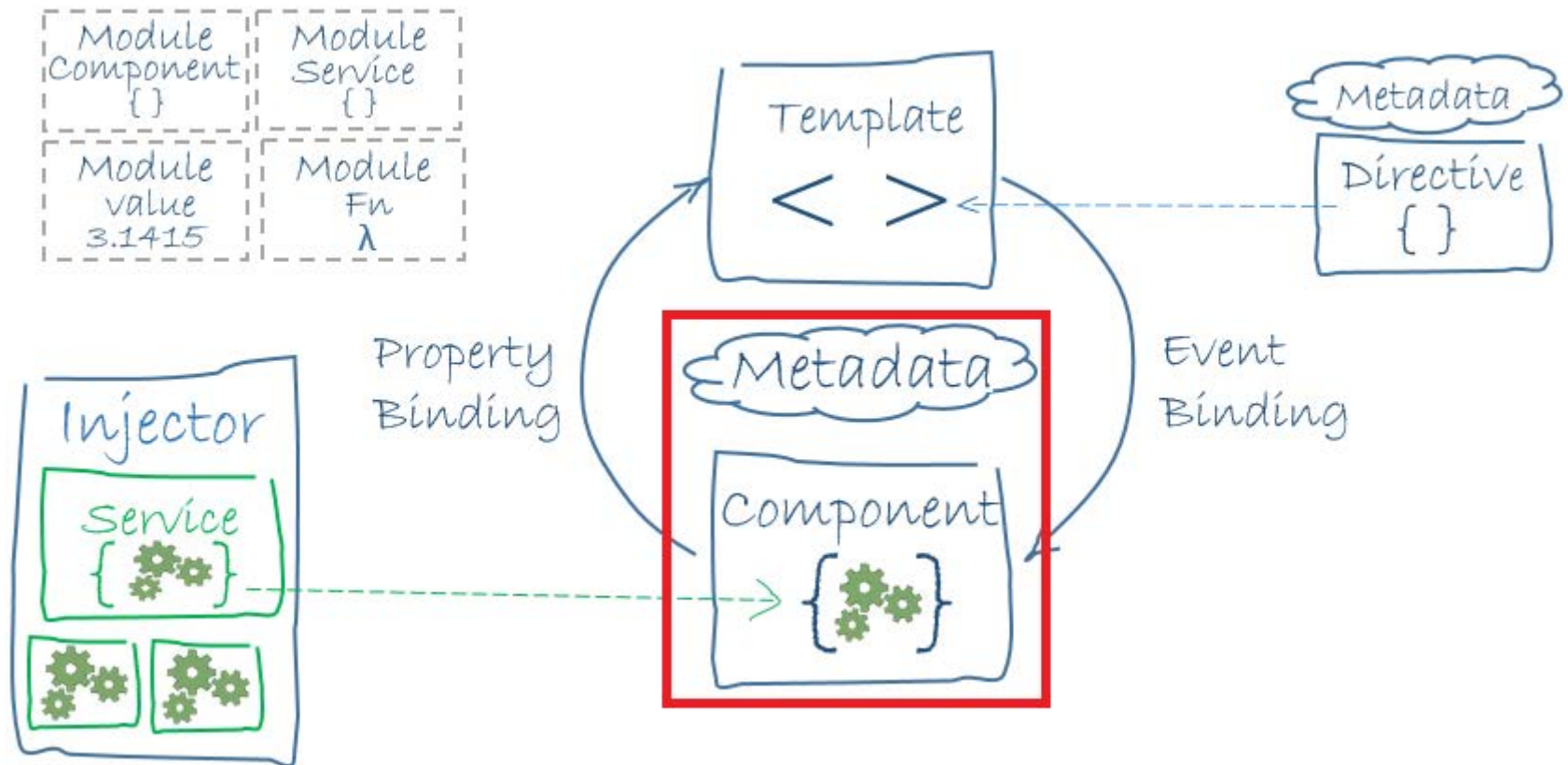
- **declarations**: specifica le *view classes* (*components*, *directives* e *pipes*) appartenenti al modulo
- **exports**: sottoinsieme delle view classes della proprietà **declarations** che indica le view classes che dovrebbero essere visibili ed utilizzabili nei componenti di altri moduli
- **imports**: specifica le classi esportate da altri moduli che sono necessarie alle classi dichiarate in questo modulo

# Principali proprietà @NgModule (2)

---

- **providers**: creatori di servizi che questo modulo aggiunge alla collezione globale di servizi e che diventano accessibili in tutte le parti dell'applicazione
- **bootstrap**: indica l'*application view* principale, che ospita tutte le altre. Solo il modulo **root** dovrebbe settare questa proprietà

# Components e Metadata



# Components e Metadata

---

- I ***Component*** sono le unità di base delle UI delle applicazioni Angular. Un'applicazione è un albero di componenti Angular
- Un Component è una classe adornata con il decoratore **@Component**
- Questa classe rende disponibili dei dati al template e gestisce la logica di interazione con l'utente

# Proprietà principali @Component

---

- **selector**: selettore CSS che identifica il componente in un template
- **template**: template, definito inline, della vista
- **templateUrl**: url di un file esterno che contiene il template della vista
- **style**: stile, definito inline, da applicare alla vista;
- **styleUrls**: url di un file esterno che contiene lo stile da applicare alla vista.

# Styles e StyleUrls

---

- Cosa succede se specifico sia la proprietà **styles** che **styleUrls**?

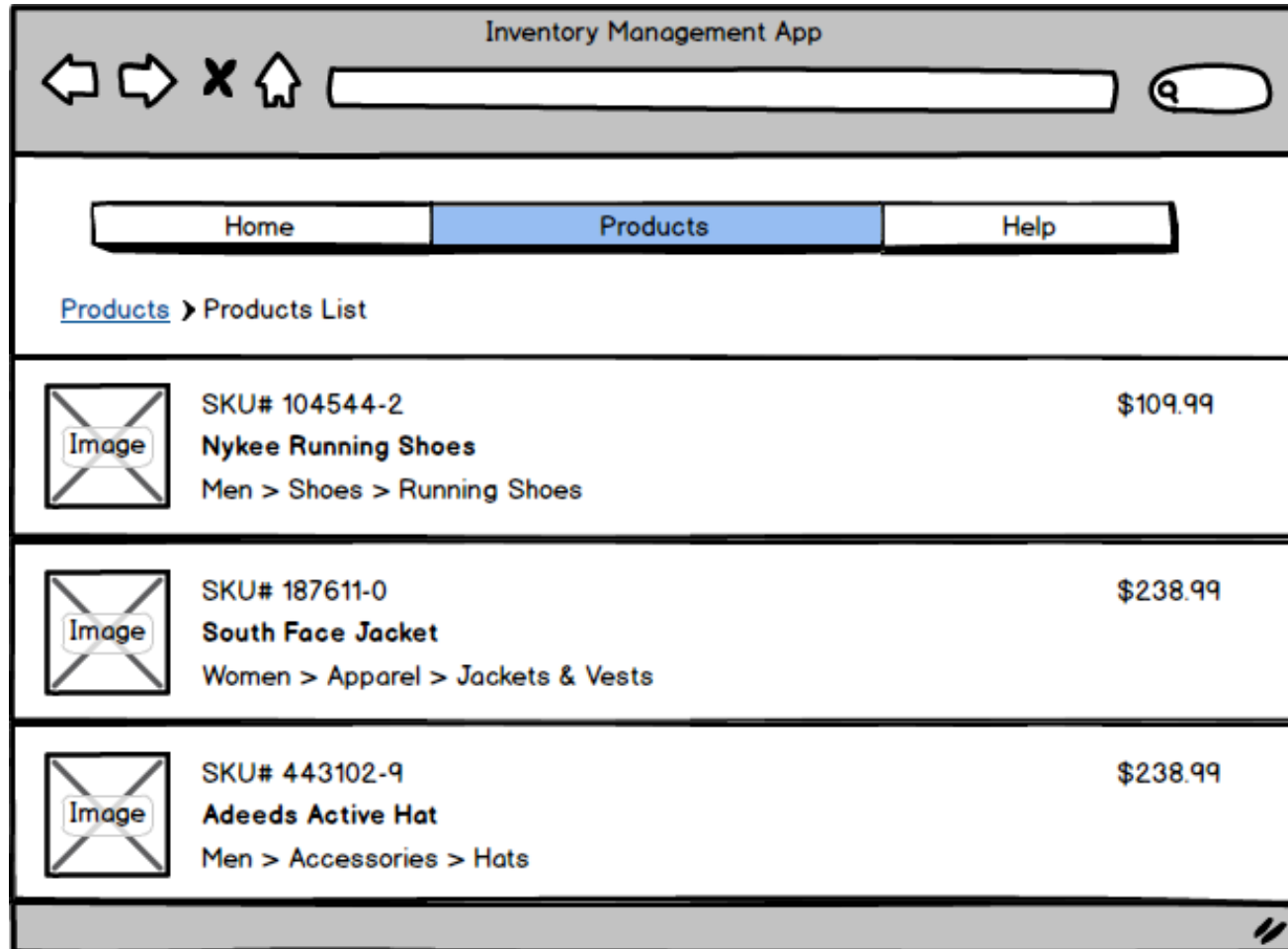
# Styles e StyleUrls

---

- Cosa succede se specifico sia la proprietà **Styles** che **StyleUrls**?
  - Non esiste una priorità sulle regole. Viene semplicemente utilizzato lo stile specificato nell'ultima proprietà definita



# Inventory Management App



# Components

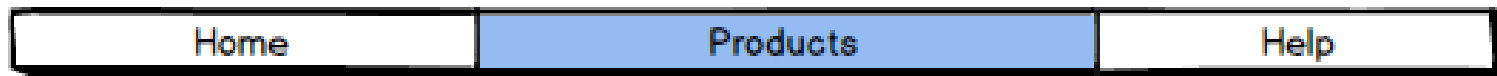
---

- Dato questo mockup, per prima cosa si possono suddividere gli elementi in componenti di alto livello
- Si possono identificare 3 componenti di alto livello:
  - *Navigation Component*
  - *Breadcrumbs Component*
  - *Product List Component*

# Navigation e Breadcrumb Component

---

- Navigation Component






- Breadcrumbs Component

[Products](#) > Products List

# Product List Component

---

	SKU# 104544-2 <b>Nykee Running Shoes</b> Men > Shoes > Running Shoes	\$109.99
	SKU# 187611-0 <b>South Face Jacket</b> Women > Apparel > Jackets & Vests	\$238.99
	SKU# 443102-9 <b>Adeeds Active Hat</b> Men > Accessories > Hats	\$238.99

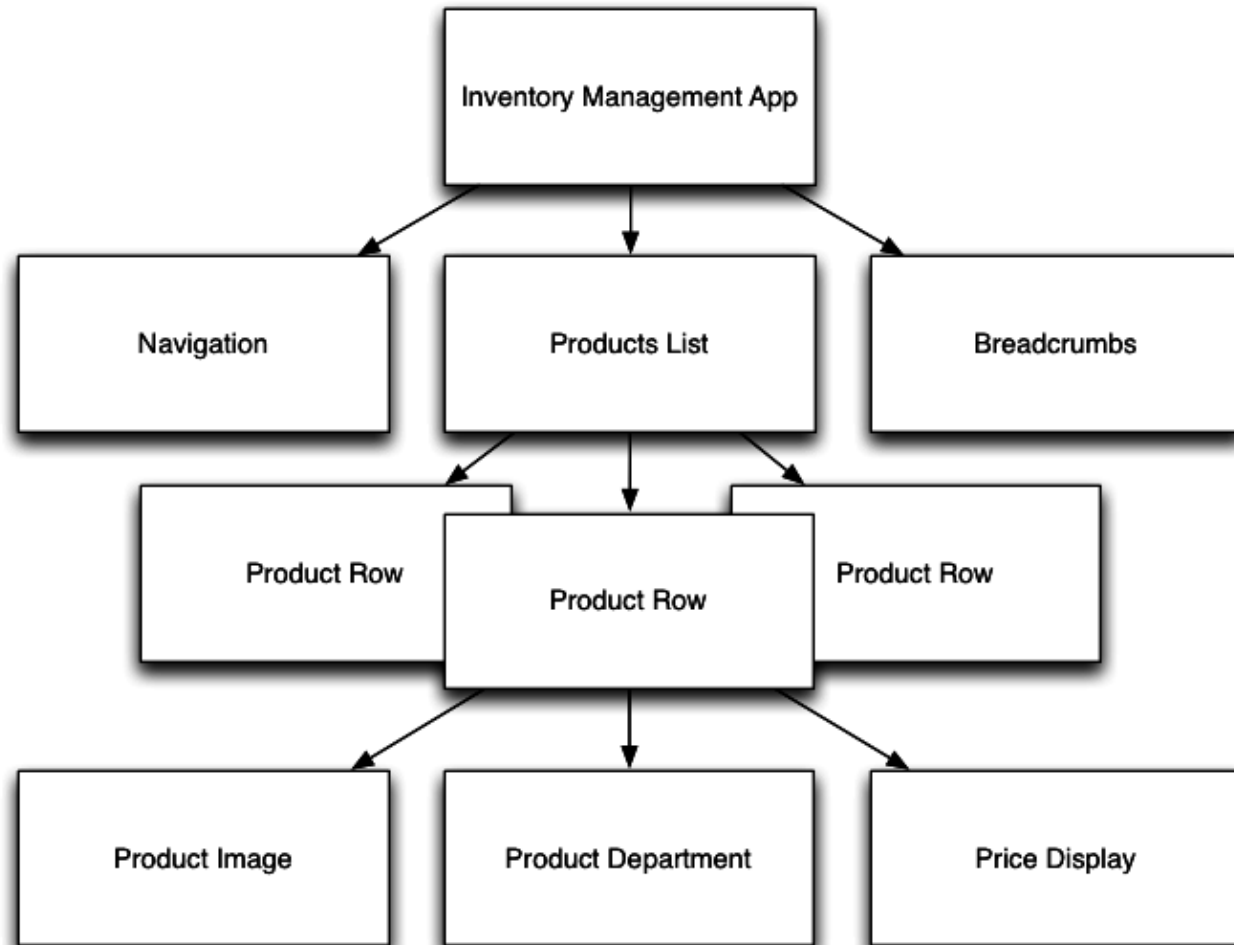
# Product Row Component

- Il componente *Product List* potrebbe essere ulteriormente diviso in componenti più piccoli, ad esempio *Product Row*



- Si potrebbe poi continuare suddividendo ogni *Product Row* in:
  - *Product Image*
  - *Product Department*
  - *Price Display*

# App Tree Diagram



# Components Lifecycle Hook

---

- I metodi «**hook**» sono eventi a cui è possibile registrarsi e che vengono chiamati da Angular durante il ciclo di vita di direttive e componenti
- Questi metodi di callback sono chiamati DOPO la chiamata al costruttore. Di seguito, sono riportati i metodi, ordinati in base all'ordine di chiamata

# Ng Lifecycle Hook

---

1. **ngOnChanges**: prima di **ngOnInit** e ogni volta che cambia una input **property** sul componente
2. **ngOnInit**: all'inizializzazione del componente
3. **ngDoCheck**: chiamato ogni volta che si verifica un ciclo di cambiamento
4. **ngAfterContentInit**: dopo la “proiezione” dei contenuti provenienti dall'esterno nella vista

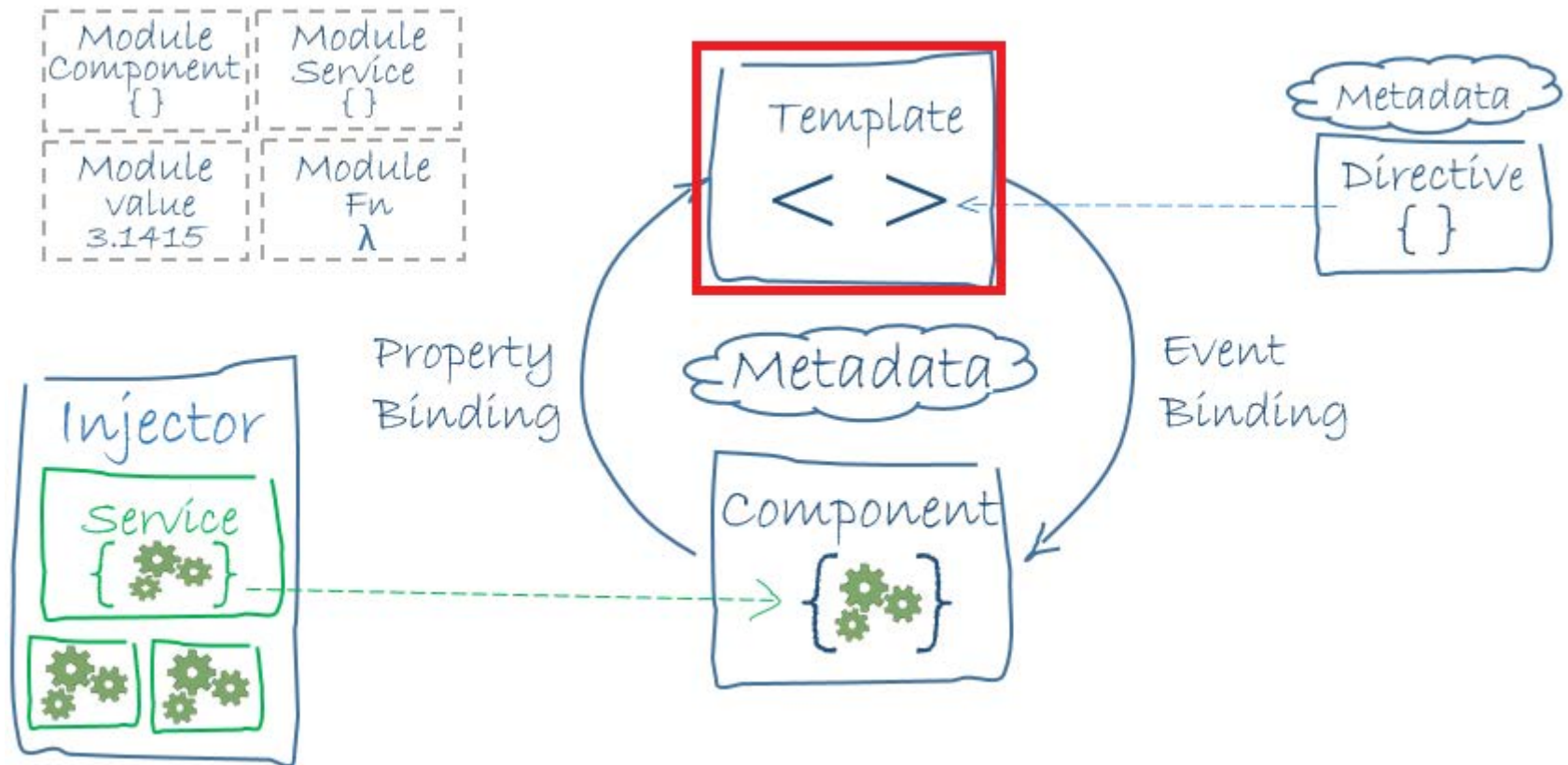


# Ng Lifecycle Hook (2)

---

- 5. **ngAfterContentChecked**: dopo il controllo dei contenuti provenienti dall'esterno su cui si è effettuato il binding
- 6. **ngAfterViewInit**: al termine della renderizzazione della vista (compresi i figli)
- 7. **ngAfterViewChecked**: dopo il controllo dei binding sulle viste (compresi i figli)
- 8. **ngOnDestroy**: prima della distruzione del componente. Questo è il posto giusto dove fare gli unsubscribe degli *Observable*, per evitare memory leak

# Template

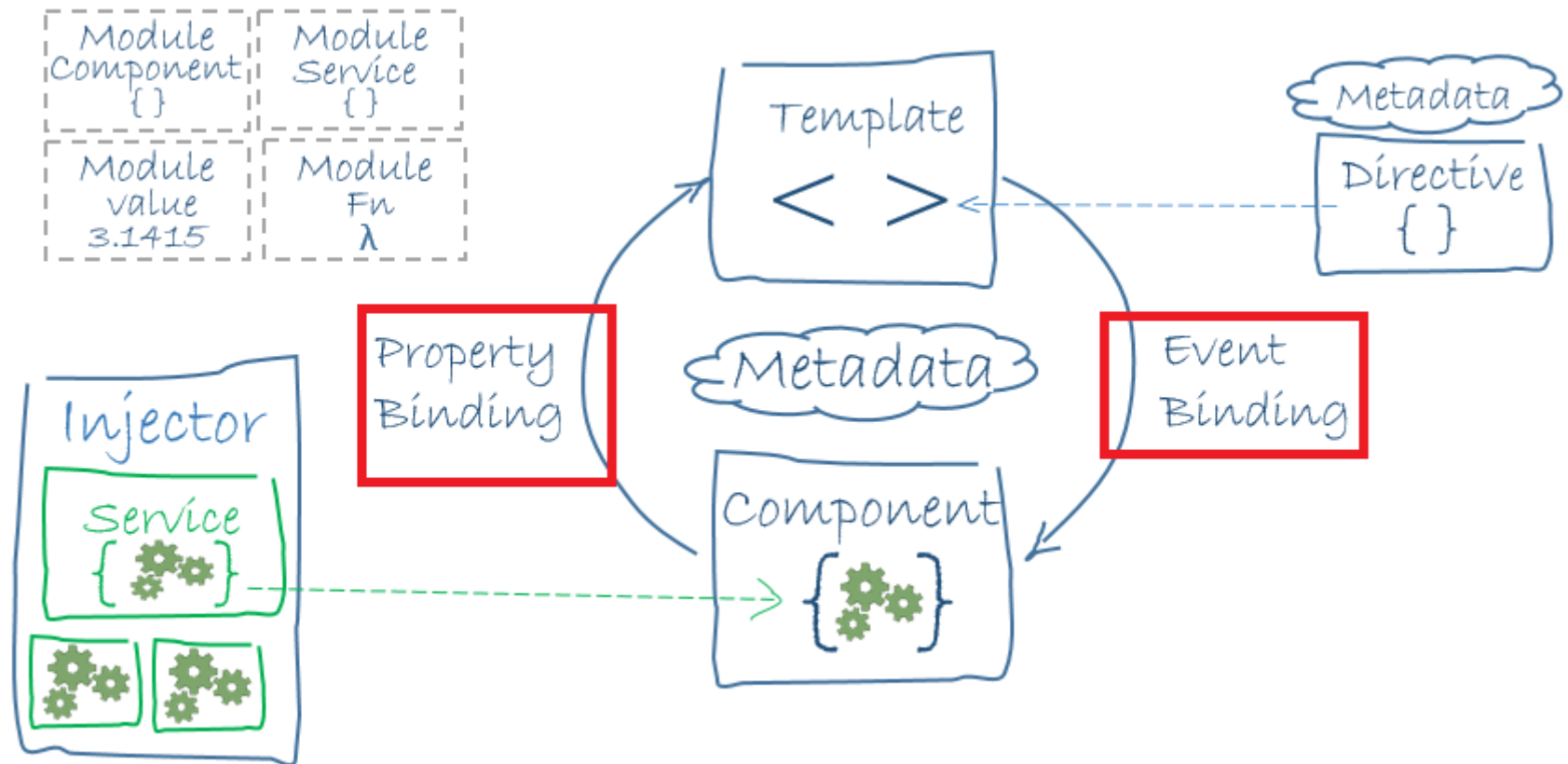


# Templates

---

- Rappresenta la vista del componente, ovvero come deve essere renderizzato
- È composto da:
  - Tag HTML;
  - Sintassi di Angular per il template

# Data Binding

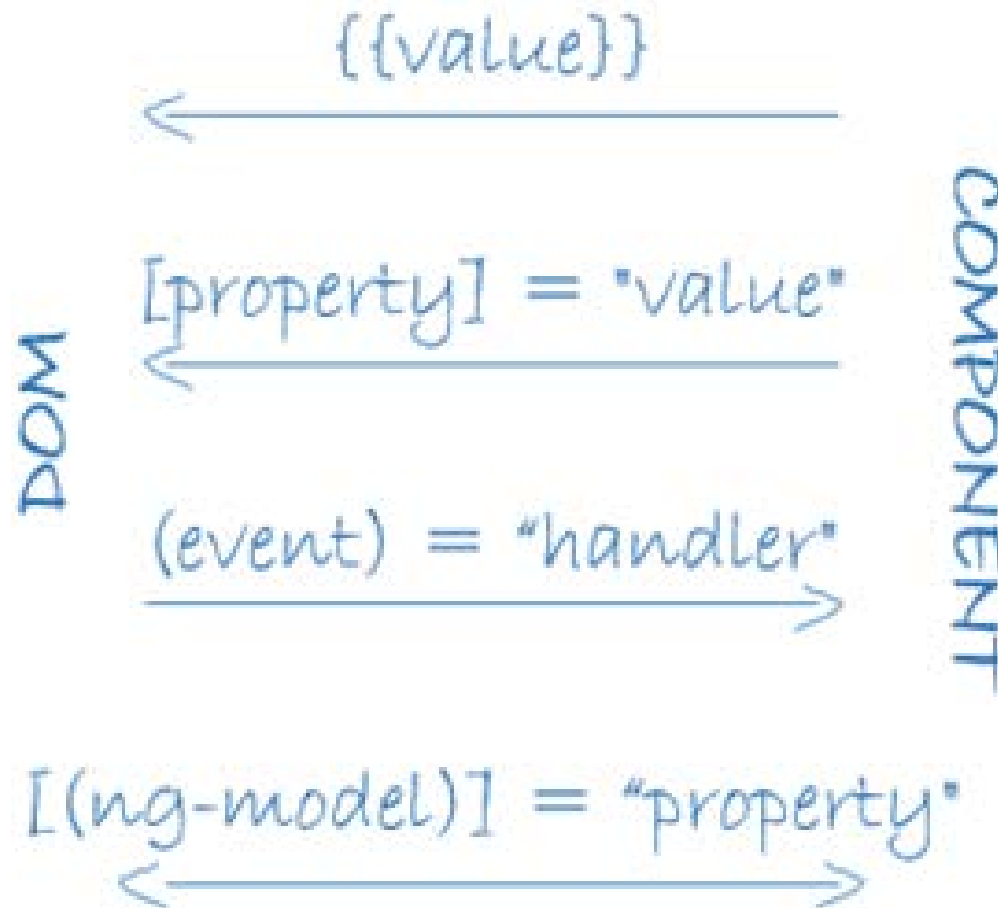


# Data Binding

---

- Angular supporta il data binding, un meccanismo per coordinare la comunicazione tra un componente e il suo template
- Aggiungendo il markup per eseguire il binding al template HTML si specifica ad Angular come connettere il DOM e il Component

# Angular Data Binding



# Interpolation

---

 `{{value}}`

- L'espressione contenuta dalle parentesi graffe, viene valutata e infine convertita in una stringa
- Sono proibite espressioni che hanno side effect (assegnamenti, ...)
- Le espressioni possono contenere proprietà del Component che controlla la vista

# Property Binding

---

`[property] = "value"`



- L'espressione contenuta tra i doppi apici viene valutata e assegnata all'attributo di un elemento del DOM
- Per quanto riguarda l'espressione, valgono le stesse considerazioni fatte per l'interpolazione



# Interpolation vs Property Binding

- Primo esempio:

- Interpolazione

<p>

`` is the  
`<i>interpolated</i>` image.

</p>

- Property Binding

<p>

`<img [src]="heroImageUrl">` is the  
`<i>property bound</i>` image.

</p>

# Interpolation vs Property Binding

---

- Secondo esempio:

- Interpolazione

`<p>`

`"<span>{{title}}</span>" is the  
<i>interpolated</i> title.`

`</p>`

- Property Binding?

# Interpolation vs Property Binding

- Secondo esempio:

- Interpolazione

`<p>`

`"<span>{{title}}</span>" is the  
<i>interpolated</i> title.`

`</p>`

- Property Binding?

`<p>`

`"<span [innerHTML]="title"></span>"  
is the <i>property bound</i> title.`

`</p>`

# Interpolation o Property Binding?

---

- Angular traduce le interpolazioni nella corrispondente *property binding* prima di renderizzare la vista
- Non ci sono motivi tecnici per preferire una forma all'altra. Considerando la leggibilità, si tende a preferire l'interpolazione

# Event Binding

---

(event) = "handler" →

- L'*Event Binding* di Angular consente di definire un handler per un determinato evento, ad esempio per i movimenti del mouse o i click
- Esempio:  
`<button (click)="onSave()">Save</button>`

# Two-way Data Binding

`[(ng-model)] = "property"`  


- Il *Two-way Data Binding* è l'unione del *property* e dell'*event binding*. In questo modo il valore fluisce dal DOM al component e viceversa
- Per il Two-way Data Binding si usa la sintassi "*banana in a box*"
- Esempio:

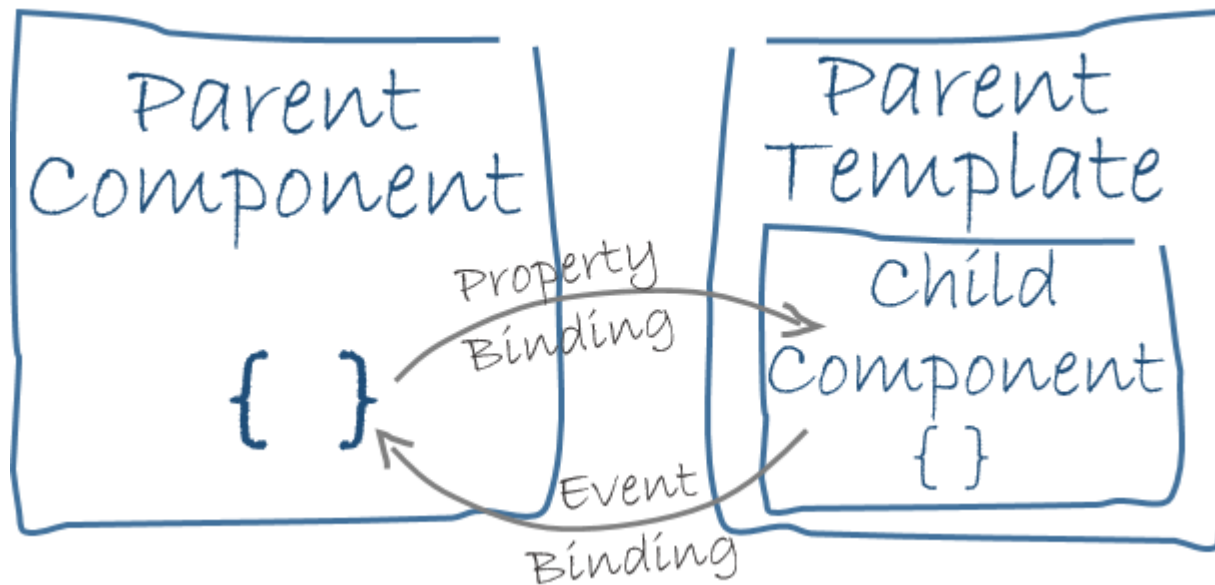
```
<input [(ngModel)]="element" />
```

```
<input [ngModel]="element"
```

```
(ngModelChange)="element = $event" />
```

# Data Binding tra Component

- Il data binding è anche importante per la comunicazione tra component padre e figlio



# Pipes

- Le *Pipe* sono un modo di scrivere trasformazioni di un valore direttamente nel template
- Sintassi:  
`<p>Testo {{ value | pipe }}</p>`
- È possibile parametrizzare le pipe  
`<p>  
 Today is {{ curdate | date:"dd/MM/yy" }}  
</p>`
- È possibile concatenare le pipe

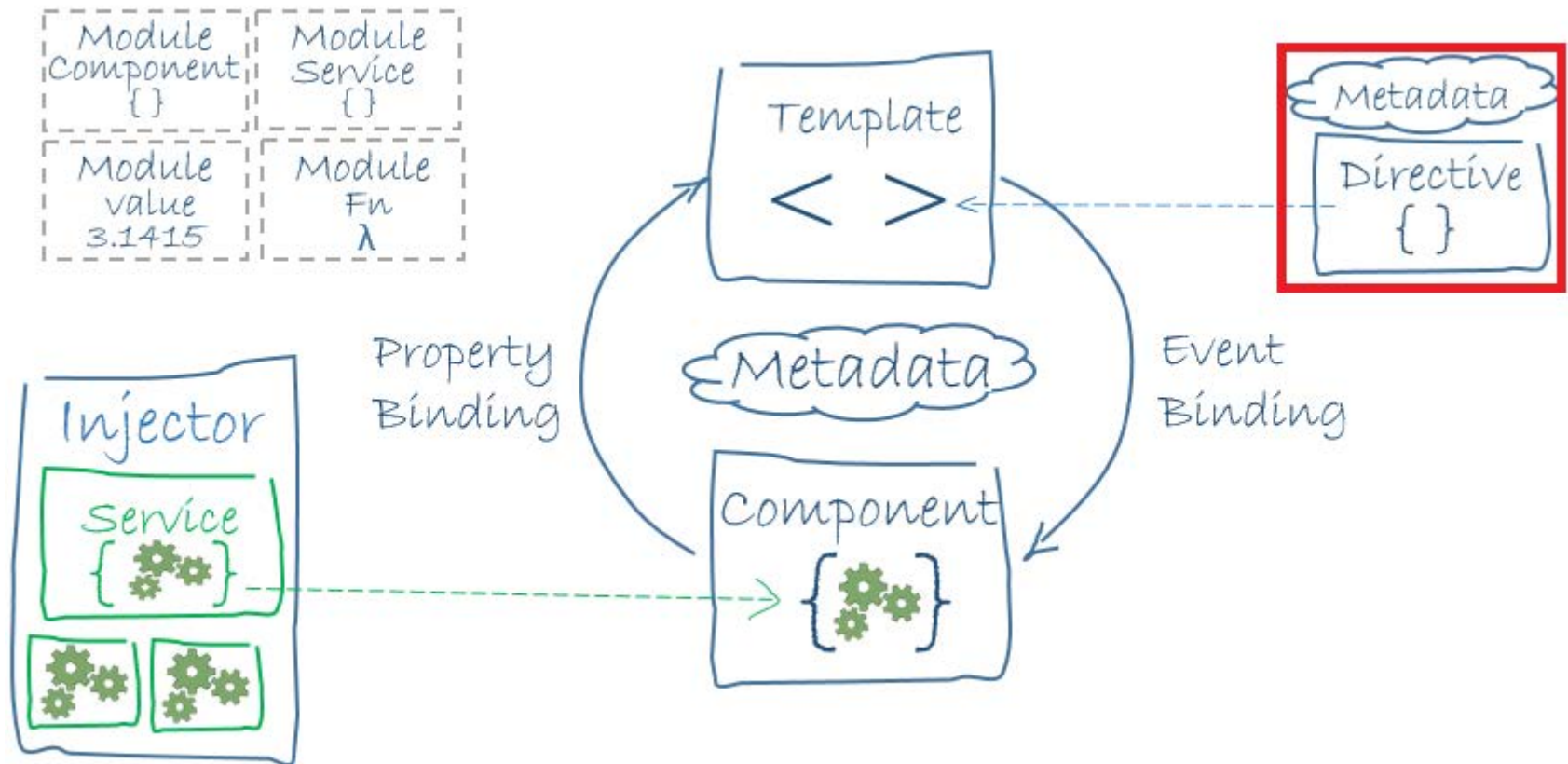


# Pipe Built-in

---

- Esistono Pipe built-in, ad esempio:
  - *DatePipe*
  - *UpperCasePipe*
  - *CurrencyPipe*
  - *PercentPipe*
- È possibile creare pipe personalizzate utilizzando il decoratore **@Pipe**

# Directive



# Directives

---

- Ci sono tre tipi di direttive in Angular:
  - **Components**: sono direttive che hanno un template
  - ***Direttive strutturali***: sono direttive che cambiano il layout del DOM aggiungendo o rimuovendo elementi
  - ***Direttive dell'attributo***: sono direttive che cambiano l'aspetto o il comportamento di un elemento, componente o di un'altra direttiva

# Structural Directives

---

- Si applicano ad un elemento «**host**». La direttiva fa quindi tutto quello che dovrebbe fare con l'elemento **host** e con i suoi discendenti
- Sono semplici da riconoscere in quanto, il loro nome è preceduto da un asterisco
- Esempio:

```
<div *ngIf="hero" >{{hero.name}}</div>
```

# Built-in Structural Directives

---

- Le direttive strutturali built-in sono:
  - `ngIf`
  - `ngSwitch`
  - `ngFor`
- È possibile creare direttive strutturali personalizzate

# ngIf

---

- Consente di visualizzare o meno un elemento, in base ad una condizione
- La condizione è determinata dal risultato dell'espressione passata nella direttiva
- Se il risultato dell'espressione è un valore *falsy*, l'elemento sarà rimosso dal DOM

# Esempi ngIf

---

- `<div *ngIf="false"></div>`
- `<div *ngIf="a > b"></div>`
- `<div *ngIf="str == 'yes' "></div>`
- `<div *ngIf="myFunc( ) "></div>`

# Esempi ngIf

- `<div *ngIf="false"></div>`  
`<!-- mai visualizzato-->`
- `<div *ngIf="a > b"></div>`  
`<!-- visualizzato se a è maggiore di b -->`
- `<div *ngIf="str == 'yes'"></div>`  
`<!-- visualizzato se str è la stringa "yes" -->`
- `<div *ngIf="myFunc()"></div>`  
`<!-- visualizzato se myFunc ritorna truthy -->`



# ngSwitch

---

- Consente di visualizzare o meno diversi elementi, in base ad una condizione
- Viene utilizzata insieme a:
  - **ngSwitchCase**: descrive una condizione definita
  - **ngSwitchDefault**: gestisce tutti i restanti casi non definiti

# Esempio ngSwitch

---

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchCase="'A'">Var is A</div>
  <div *ngSwitchCase="'B'">Var is B</div>
  <div *ngSwitchCase="'C'">Var is C</div>
  <div *ngSwitchDefault>
    Var is something else
  </div>
</div>
```

# ngFor

---

- Questa direttiva ripete l'elemento del DOM (o una collezione di elementi)
- Esempio:  

```
cities = ['Miami', 'Sao Paulo', 'New York'];  
<ul *ngFor="let city of cities">  
  <li>{{city}}</li>  
</ul>
```
- Nell'esempio, l'array **cities** conteneva stringhe ma è possibile usare la direttiva **ngFor** anche con array contenenti oggetti o array

# ngFor sintassi estesa

---

- **\*ngFor="let item of items; index as i; trackBy: trackByFn"**
- Valori esportati:
  - **index**: number che rappresenta l'indice del item corrente
  - **first**: booleano settato a True quando l'elemento è il primo
  - **last**: booleano settato a True quando l'elemento è l'ultimo
  - **even**: booleano settato a True quando l'elemento ha un indice pari
  - **odd**: booleano settato a True quando l'elemento ha un indice dispari

# Esempio ngFor

---

```
cities = ['Miami', 'Sao Paulo', 'New York'];  
<ul *ngFor="let city of cities; index as I;  
odd as isOdd">  
  <li [ngClass]="{odd: isOdd}">  
    {{i}} - {{city}}  
  </li>  
</ul>
```

# Attribute Directives

---

- Idealmente, una direttiva dovrebbe lavorare in modo da essere «*component agnostic*» e non legata ai dettagli implementativi
- Le direttive dell'attributo built-in sono:
  - `ngClass`
  - `ngNonBindable`
  - `ngStyle`
- È possibile creare direttive dell'attributo personalizzate

# ngClass

- Consente di settare o cambiare le classi Css di un dato element del DOM
- Esempi:

```
<div [ngClass]="{bordered: false}">
```

```
  This is never bordered</div>
```

```
<div [ngClass]="{bordered: true}">
```

```
  This is always bordered</div>
```

```
<div [ngClass]="{bordered: isB}">
```

```
  Using object literal. Border  
  {{ isB ? "ON" : "OFF" }}
```

```
</div>
```

# ngNonBindable

---

- Consente di specificare una particolare sezione della pagina che non sarà compilata o su cui non sarà eseguito il binding

- Esempio:

```
<span class="pre" ngNonBindable>  
    Modo per stampare {{ variablename }}  
</span>
```



# ngStyle

---

- Consente di settare, ad un elemento del DOM, una proprietà CSS
- Esempio

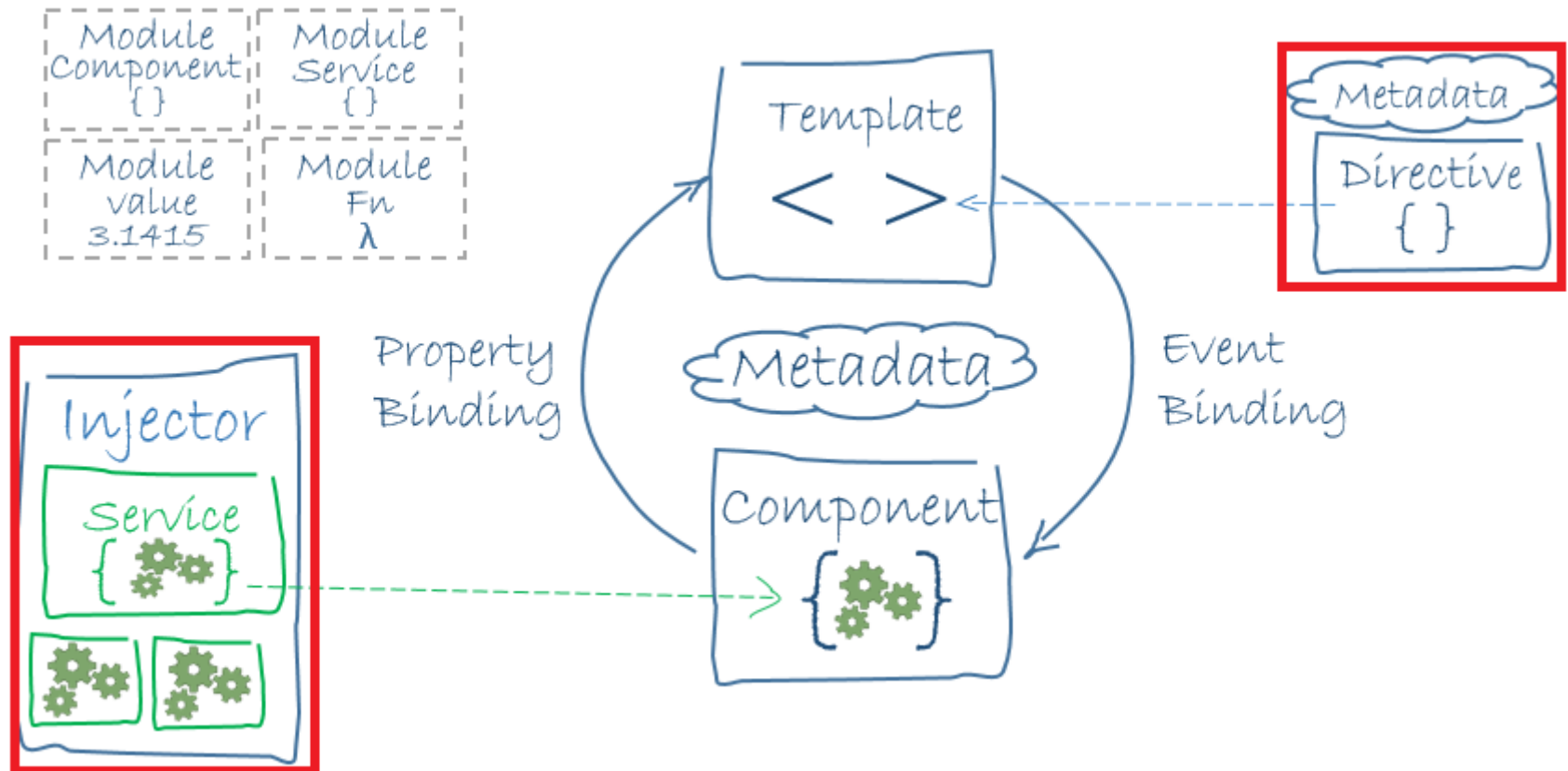
```
<span [ngStyle]="{color: color}">
    {{ color }} text
</span>
```

# `<ng-container>`

---

- `<ng-container>` è un elemento di raggruppamento che non interferisce con lo stile o con il layout perché Angular non lo aggiunge al DOM
- Può essere usato in tutti quei casi in cui non si ha a disposizione un elemento “contenitore” o quando si vuole applicare una direttiva a semplice testo

# Services



# Services

---

- I servizi sono una vasta categoria che comprende qualsiasi valore, funzione o caratteristica di cui l'applicazione ha bisogno
- Un servizio è tipicamente una classe con uno scopo ben definito (ad esempio logging o caricamento di dati)
- Angular non ha una definizione di un servizio e non prevede una classe base o un posto in cui registrarli

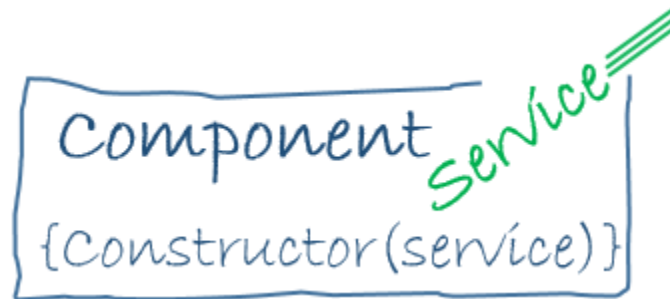
# Components vs Services

---

- Il compito del *Component* è di «gestire» la User eXperience
- Un *Component* non dovrebbe caricare i dati dal server o validare l'input di un utente. Questi compiti devono essere delegati ai *servizi*

# Dependency injection

- La ***Dependency injection*** è un modo di fornire una nuova istanza di una classe con le dipendenze che richiede pienamente formate
- Molte dipendenze sono servizi
- Angular stabilisce di quali servizi un component necessita guardando i tipi dei parametri del costruttore



A hand-drawn diagram illustrating a component's constructor. It consists of a blue rectangular box. Inside the box, the word "Component" is written in blue. Below it, the text "{Constructor(service)}" is written in blue. To the right of the box, the word "Service" is written in green, with three green lines extending from its top right corner, suggesting a dependency or injection.

# Dependency Injection

---

- Quando Angular crea un componente, chiede all'**injector** i servizi richiesti
- Un injector mantiene un container delle istanze dei servizi che sono stati precedentemente creati. Se un'istanza di un servizio richiesto non è nel **container**, l'injector ne crea uno e lo aggiunge al container
- Quando tutti i servizi richiesti sono stati risolti e restituiti, Angular chiama il costruttore con quei servizi come argomenti

# Dependency Injection Gerarchica

---

- Il sistema di Dependency Injection è gerarchico e segue la gerarchia dei componenti
- È come se ogni componente avesse a disposizione un “*injector*” al quale chiedere se esiste un’istanza del servizio o un provider per istanziarlo: se non esiste, l’injector chiede la dipendenza all’injector del componente padre, risalendo la gerarchia



# Dichiarazione di un servizio

---

- Un servizio si può dichiarare:
  - A livello di bootstrap dell'applicazione: tutti i componenti avranno la stessa istanza dei servizi
  - A livello di componente: l'istanza sarà condivisa solo ed esclusivamente dal componente in cui è stato dichiarato il provider e dai suoi componenti figli

# Forms

---

- Angular mette a disposizione i seguenti strumenti per la gestione di form:
  - **FormControl**: incapsula in un oggetto l'input del form e il suo stato (valido, sporco, con errori)
  - **FormGroup**: wrapper di una collezione di FormControl
  - **Validator**: permettono di validare l'input
  - **Observer**: consentono di rilevare cambiamenti e reagire di conseguenza

# HTTP

---

- Angular ha una propria libreria HTTP che consente di chiamare API esterne in modo asincrono
- La libreria HTTP mette a disposizione i diversi metodi `http`:
  - `get`
  - `post`
  - `put`
  - Ecc...

# HTTP

---

- I tre approcci principali alla programmazione asincrona sono:
  - **Callbacks**
  - **Promises**
  - **Observables**
- L'approccio scelto da Angular sono gli **Observables**, tanto che i metodi di HTTP restituiscono degli Observables
- Angular incorpora **RxJS** e lo usa internamente

# Routing

---

- Angular ha un modulo **Router** per la navigazione. Per navigare più pagine è necessario:
  - Inserire il tag html base nella pagina principale
  - Definire le rotte come array di oggetti che mappano i path ai component
  - Inserire la direttiva **router-outlet** nella view principale
  - Definire dei link utilizzando l'attributo **routerLink**

# Data Architecture

---

- La gestione dei dati è uno degli aspetti più «tricky» per lo sviluppo di un'applicazione
- Ci sono molti modi per ottenere dati con Angular:
  - *Richieste AJAX*
  - *Websockets*
  - *Local storage*
  - *Service Workers*
  - Ecc ...

# Data Architecture

---

- Angular è estremamente flessibile riguardo la data architecture. Angular non consiglia un'architettura particolare ma cerca di rendere facile l'uso dell'architettura scelta
- Data Architecture più utilizzate:
  - ***MVC/Two-way data binding***
  - ***Flux***: pattern basato su un flusso di dati unidirezionale
  - ***Observables***: ci si iscrive a stream di dati e si eseguono azioni in reazione a cambiamenti

# AngularJS e Angular

---

- Nonostante le differenze tra AngularJS e le versioni successive esistono diversi costrutti per garantire l'interoperabilità tra le applicazioni delle diverse versioni
- In particolare, è possibile creare applicazioni ibride, che consentono di eseguire contemporaneamente AngularJS e Angular
- Questo è molto utile per il processo di aggiornamento di applicazioni basate su AngularJS, che possono essere incrementalmente aggiornate alla nuova versione di Angular



# NativeScript

---

- NativeScript è un framework per lo sviluppo cross platform di applicazioni mobile che si basa su Javascript, TypeScript o Angular
- Le applicazioni sviluppate con NativeScript, come quelle sviluppate con React Native e Xamarin, non renderizzano una web view ma utilizzano i componenti UI nativi, migliorando le performance dell'applicazione

# Style Guide

---

- Nella documentazione di Angular è presente una Style Guide che regola diversi aspetti come:
  - Sintassi
  - Convenzioni
  - Struttura dell'applicazione
- Presenta un insieme di raccomandazioni, divise in:
  - **Do**: dovrebbe essere sempre seguita
  - **Consider**: dovrebbe generalmente essere seguita
  - **Avoid**: indica qualcosa che non dovrebbe mai essere fatto
  - **Why?**: spiega il motivo della raccomandazione precedente

# Riferimenti

---

- Sito ufficiale <https://angular.io>
- Documentazione ufficiale  
<https://angular.io/docs/ts/latest/guide/>
- ng-book 2, The Complete Book on Angular