



Tecnologie e applicazioni web

Angular

Filippo Bergamasco (filippo.bergamasco@unive.it)

<http://www.dais.unive.it/~bergamasco/>

DAIS - Università Ca' Foscari di Venezia

Anno accademico: 2017/2018

Angular

Angular (conosciuto anche come Angular 2 o Angular 5) è un framework front-end per lo sviluppo di client web SPA

- Sviluppato da Google (Angular team)
- Open-source
- Basato su TypeScript
- Modulare



AngularJs vs. Angular

Angular (versione 2) è stato riscritto quasi interamente a partire dal vecchio framework web Angular 1.x. La versione stabile è stata rilasciata il 14 Settembre 2016.

Angular 1.x continua ad esistere come progetto indipendente con il nome AngularJS, ma è stato ampiamente superato dal più moderno Angular

Versioni

La versione 2, nota semplicemente come Angular, è quella che ha portato il cambiamento radicale di:

- Architettura (components vs. scope/controllers)
- Linguaggio (scritta interamente in TypeScript)
- Struttura: classi, moduli, etc (grazie a typescript)

Le versioni successive, 4 e 5 sono retro compatibili con Angular 2

Angular vs. JQuery

La filosofia alla base dei due frameworks è completamente diversa. JQuery fornisce strumenti per la manipolazione diretta del DOM.

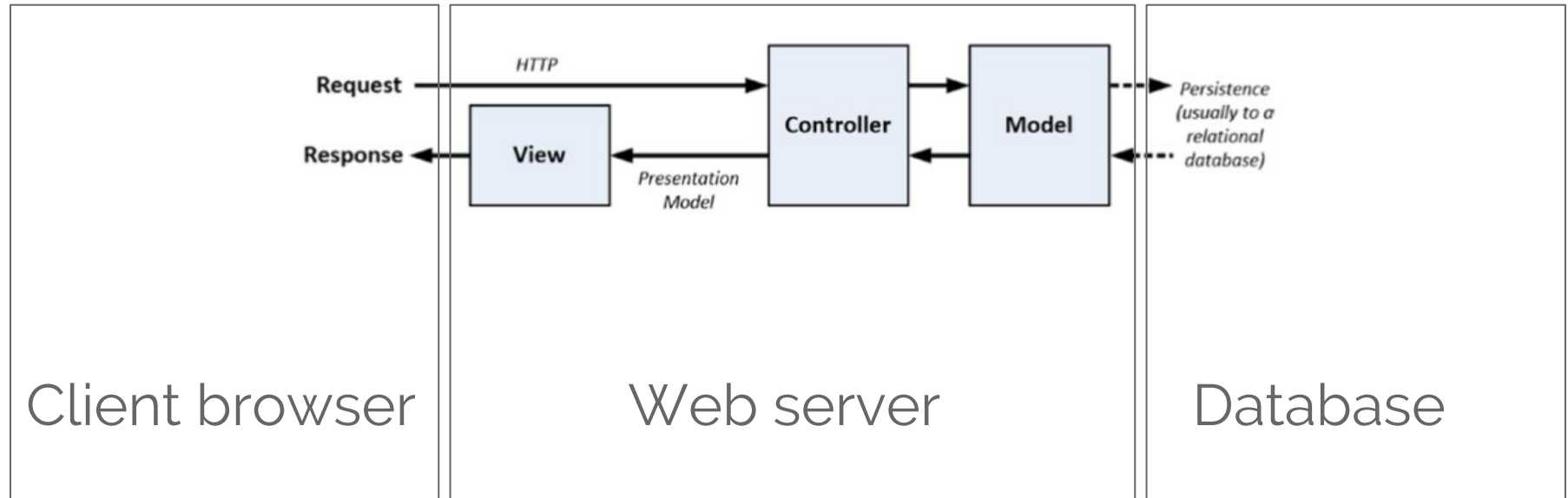
Angular è più focalizzato su una descrizione dichiarativa dei dati che vengono associati ad elementi del DOM attraverso un processo chiamato **data binding**.

Angular è un framework complesso ma con un'architettura pensata per lo sviluppo di SPA

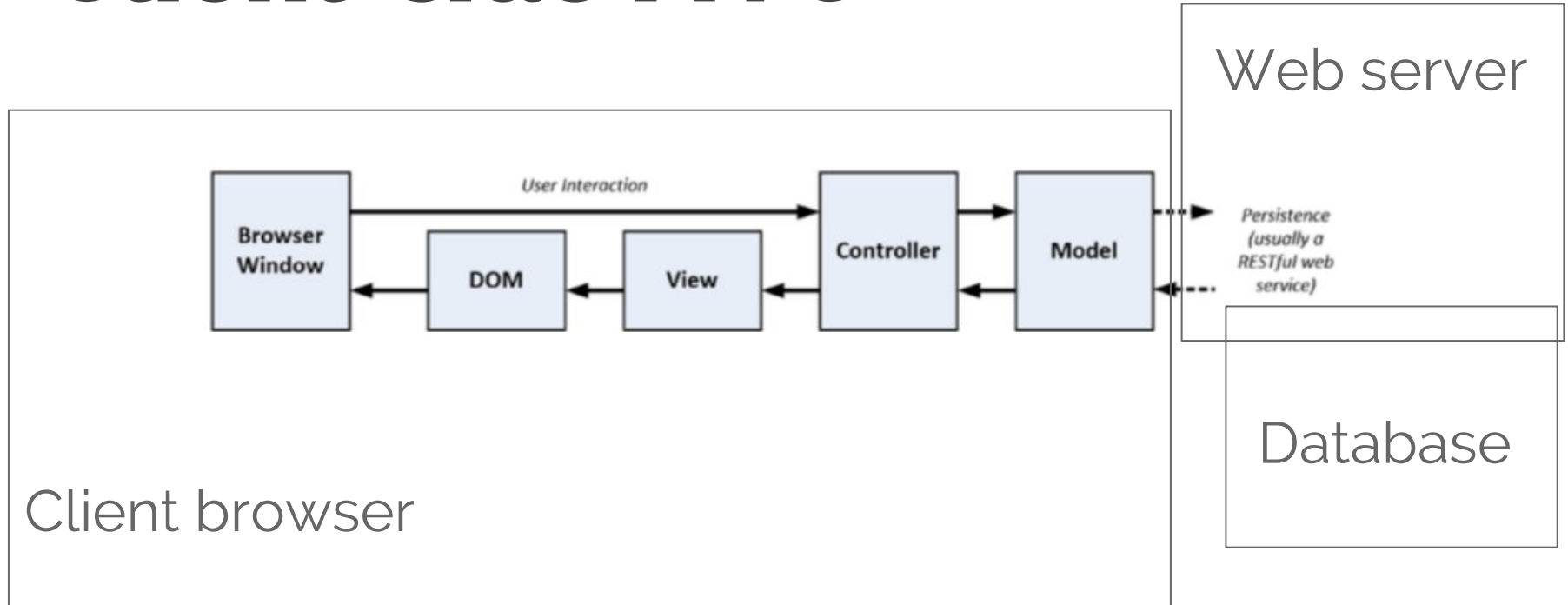
MVC da server a client

Model-view-controller è un design pattern tipicamente utilizzato per lo sviluppo di web applications perché permette la “separation of concerns” in cui il **data model** viene separato dalla **business** e **presentation** logic

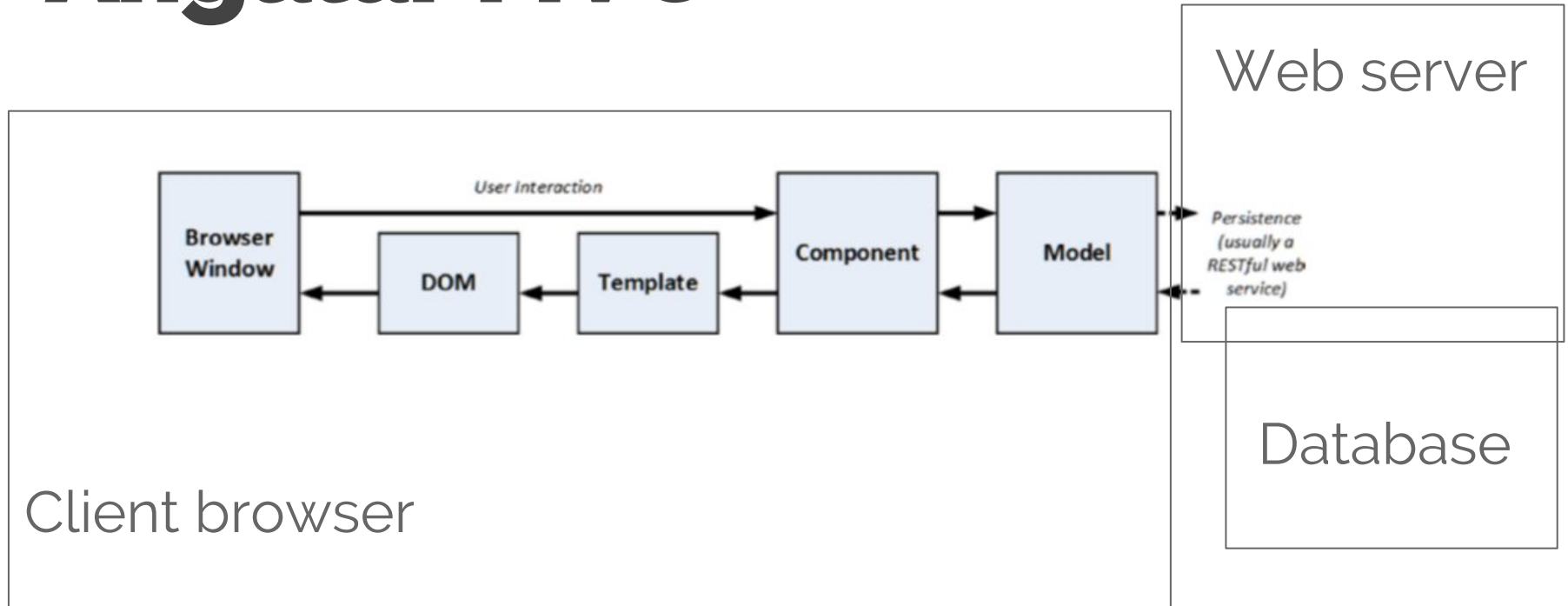
Server-side MVC



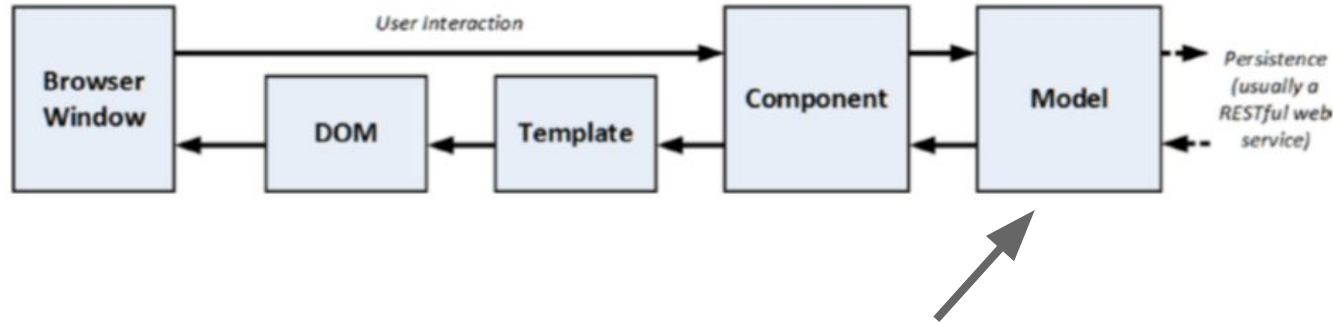
Client-side MVC



Angular MVC

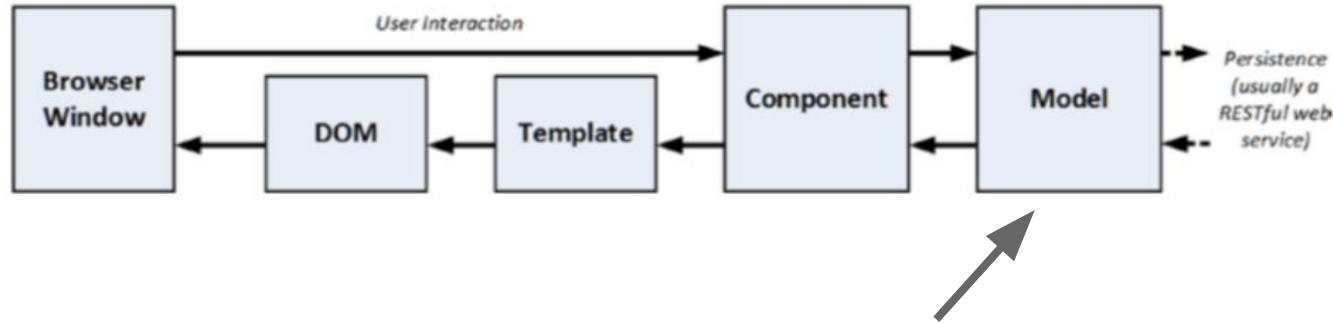


Model



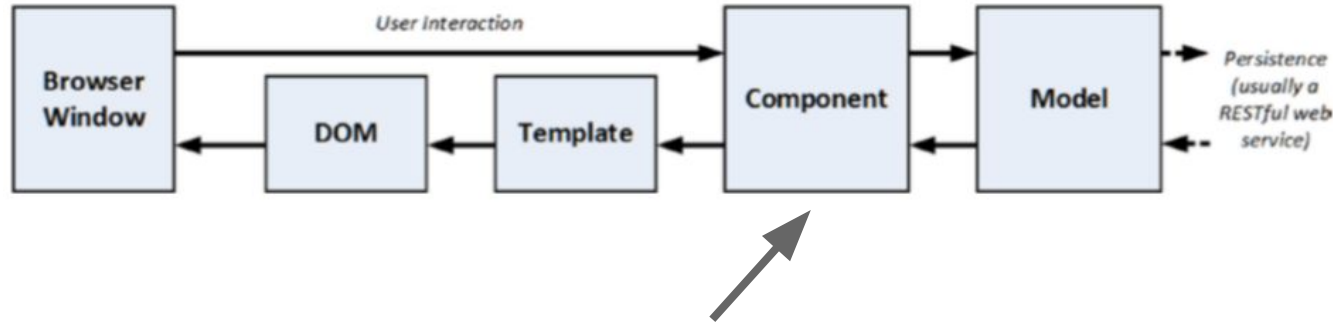
- Contiene i dati di un certo dominio
- Contiene la logica per creare, modificare e gestire i dati
- Fornisce delle API che espongono i dati e le operazioni su di essi

Model



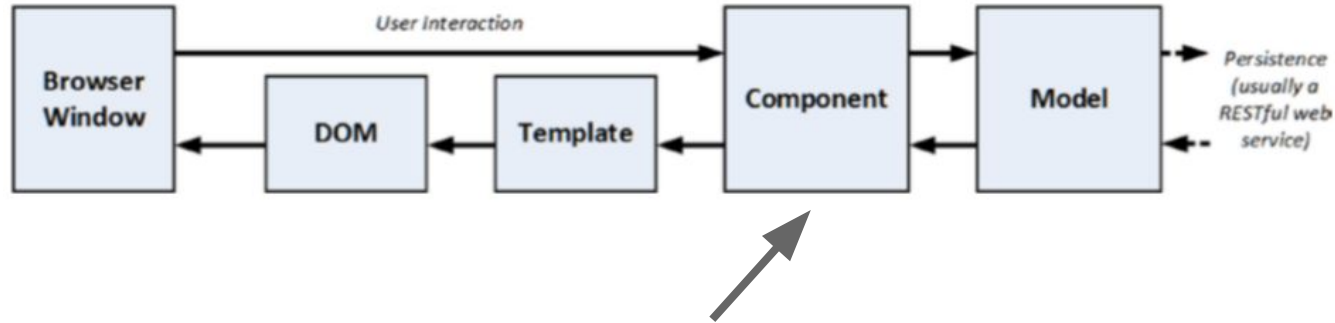
- NON deve esporre i dettagli su come i dati sono ottenuti o gestiti (il web service non deve essere esposto al controller e alle view)
- NON deve contenere la logica che trasforma i dati rispetto alle interazioni con l'utente (compito del controller)
- NON deve contenere la logica di visualizzazione dei dati (compito della view)

Component (controller)



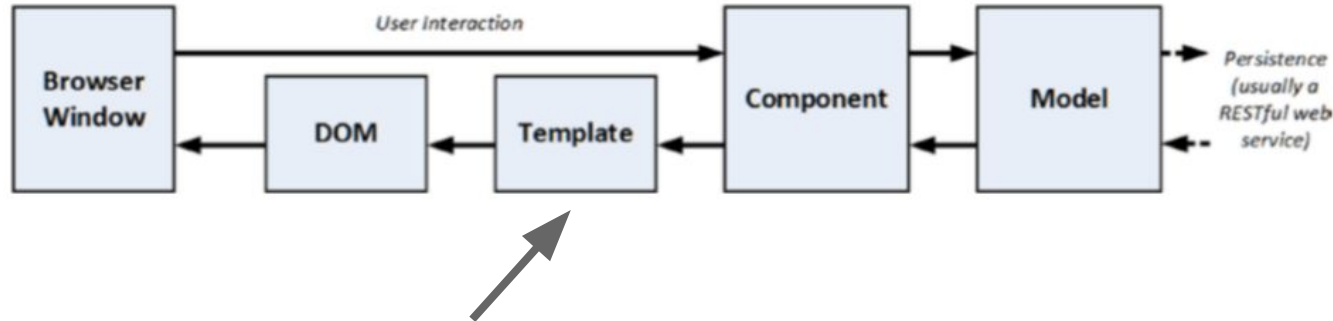
- Contiene la logica che permette di modificare il modello sulla base dell'interazione con l'utente
- Contiene la logica per impostare lo stato iniziale della view (template)
- Contiene le funzionalità richieste dal template per accedere ai dati

Component (controller)



- NON deve gestire la visualizzazione dei dati (non deve modificare il DOM)
- NON deve gestire la logica di persistenza dei dati (Compito del modello attraverso il web service)

Template (view)



- Contiene la logica di markup necessaria a presentare i dati all'utente
- NON deve contenere alcuna logica di modifica, creazione e gestione dei dati
- NON deve contenere la logica dell'applicazione

Decorators

Angular utilizza i decorator per “annotare” una classe, un metodo o delle property e aggiungere metadati e features aggiuntive a run-time.

Ad esempio un class decorator è applicato al costruttore della classe e può essere usato per osservare, modificare o sostituire la definizione della classe

Un decorator può a sua volta dipendere da parametri valutati a runtime

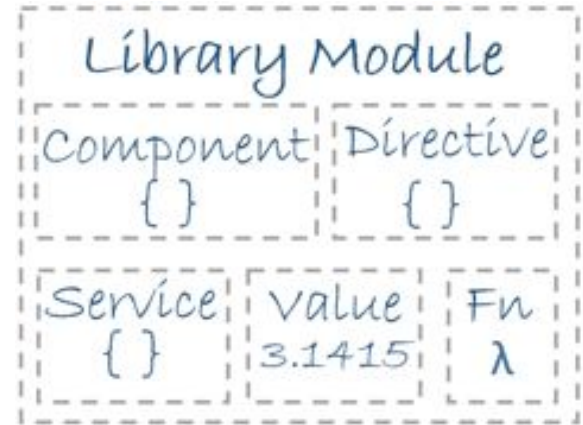
Decorators

```
function sealed(constructor: Function) {  
    Object.seal(constructor);  
    Object.seal(constructor.prototype);  
}
```

```
@sealed  
class Greeter {  
    greeting: string;  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}
```

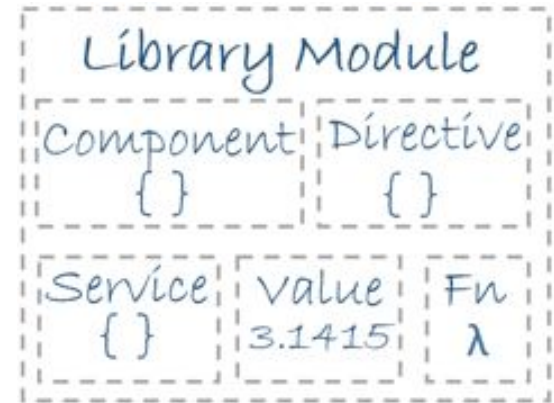

Architettura: Moduli

I building-blocks di un'app Angular sono chiamati **NgModules** che fungono da contenitori di componenti, service providers, e tutto il codice accomunato da un certo dominio, workflow o features strettamente legate



Architettura: Moduli

Ogni app Angular ha almeno un `NgModule` chiamato “root module” (per convenzione è chiamato `AppModule`) che definisce come effettuare il “bootstrap” dell’applicazione (quali componenti “root” inserire all’interno della pagina)



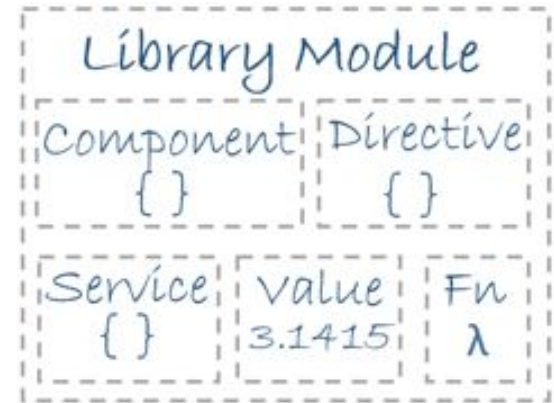
Architettura: Moduli

- Un modulo può importare ed esportare funzionalità da/verso altri moduli
- Le librerie built-in di Angular sono contenute in `NgModules`:

Ex:

`HttpModule` from `@angular/http`

`FormsModule` from `@angular/core`



Moduli

Un modulo in Angular è definito attraverso una classe decorata con il decorator `@NgModule`. Il decorator prende come parametro un oggetto che descrive i metadati del modulo:

- **Declarations:** le componenti, servizi, etc contenuti nel modulo
- **Exports:** i componenti da esportare
- **Imports:** I moduli le cui classi esportate sono necessarie ai componenti del modulo
- **Providers:** service providers esportati dal modulo

Moduli

```
import { NgModule }      from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';
```

```
@NgModule({  
  imports:      [ BrowserModule ],  
  providers:    [ Logger ],  
  declarations: [ AppComponent ],  
  exports:      [ AppComponent ],  
  bootstrap:    [ AppComponent ]  
})  
export class AppModule { }
```

Moduli Angular e Javascript

In Angular un modulo è una classe annotata con il decorator `@NgModule`.

Il browser deve però conoscere dove si trova il codice contenuto nei moduli importati. Per fare questo si utilizzano i moduli Javascript (TypeScript) che descrive l'associazione tra gli oggetti esportati e i files in cui il codice è definito

```
import { BrowserModule } from '@angular/platform-browser';
```

Moduli

```
import { NgModule } from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';
```

```
@NgModule({  
  imports: [ BrowserModule ],  
  providers: [ Logger ],  
  declarations: [ AppComponent ],  
  exports: [ AppComponent ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```



Javascript modules

Moduli

```
import { NgModule }      from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';
```

```
@NgModule({  
  imports:      [ BrowserModule ],  
  providers:    [ Logger ],  
  declarations: [ AppComponent ],  
  exports:      [ AppComponent ],  
  bootstrap:    [ AppComponent ]  
})
```

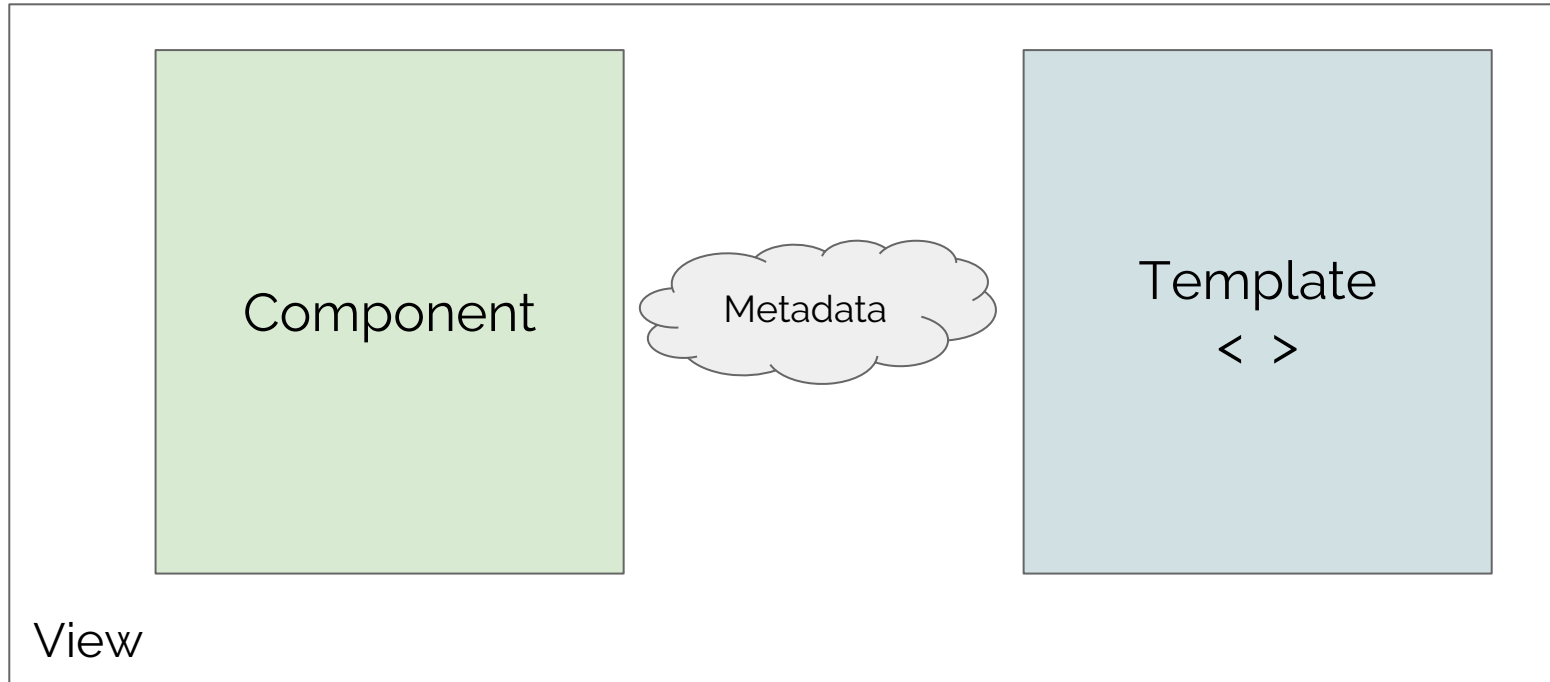
```
export class AppModule { } ← Angular module
```


Architettura: Components

I components permettono di visualizzare i dati dell'applicazione all'utente controllando una porzione di schermo visibile (DOM) chiamata "view".

Un **Component** è formato da una **classe** che contiene i dati e la logica ad essi associati. Attraverso i metadati definiti dal **@Component** decorator è associato ad un **template** HTML che definisce la view da visualizzare

Architettura: Components



Components

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

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

```
export class AppComponent {  
  title = 'app';  
}
```

app.component.ts

```
<!doctype html>  
<html lang="en">  
<head><meta charset="utf-8"><title>Testng1</title>  
  <base href="/">  
<meta name="viewport" content="width=device-width,  
initial-scale=1">  
<link rel="icon" type="image/x-icon" href="favicon.ico">  
</head>  
<body>  
  <app-root></app-root>  
</body>  
</html>
```

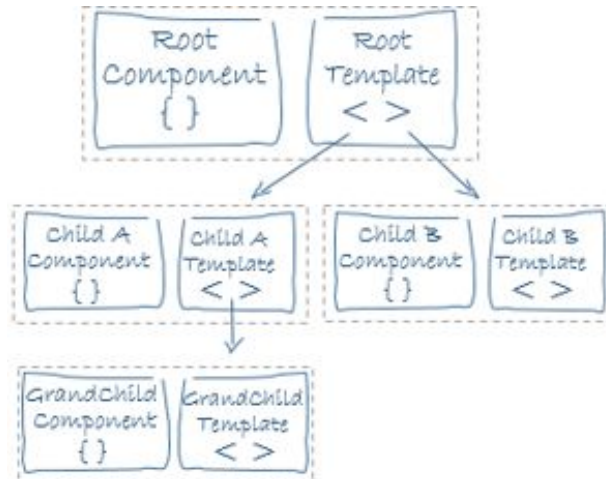
index.html

```
<div style="text-align:center">  
  <h1>  
    Welcome to {{ title }}  
  </h1>  
</div>
```

app.component.html

Gerarchia tra viste

Le views possono essere strutturate in una gerarchia (una view può includere altre view). Una child-view può appartenere allo stesso modulo o ad un altro NgModule

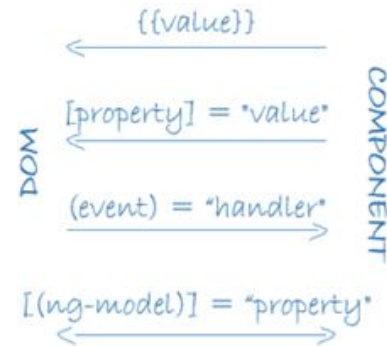


Data binding

Come sincronizzare i dati presenti nella gerarchia dei componenti con il DOM dei template?

Attraverso il meccanismo del binding markup:

- **Interpolation:** Inserisce nel codice HTML la stringa valutata in un *espressione*
- **Property binding:** Setta la proprietà di un view element (DOM o Component) sulla base di un espressione
- **Event binding:** Permette ad un componente di rispondere ad un evento del DOM



Data binding

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

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

```
export class AppComponent {  
  title = 'app';  
  isDisabled = true;  
  
  disableToggle() {  
    this.isDisabled = !this.isDisabled;  
  }  
}
```

```
<div style="text-align:center">  
  <h1>  
    Welcome to {{ title }}  
  </h1>  
  
  <button [disabled]="isDisabled" >Cancel is {{  
isDisabled ? "disabled" : "enabled" }}  
  </button>  
  
  <button (click)="disableToggle()" >  
    {{isDisabled ? "enable" : "disable"}} the  
    other button</button>  
  
</div>
```

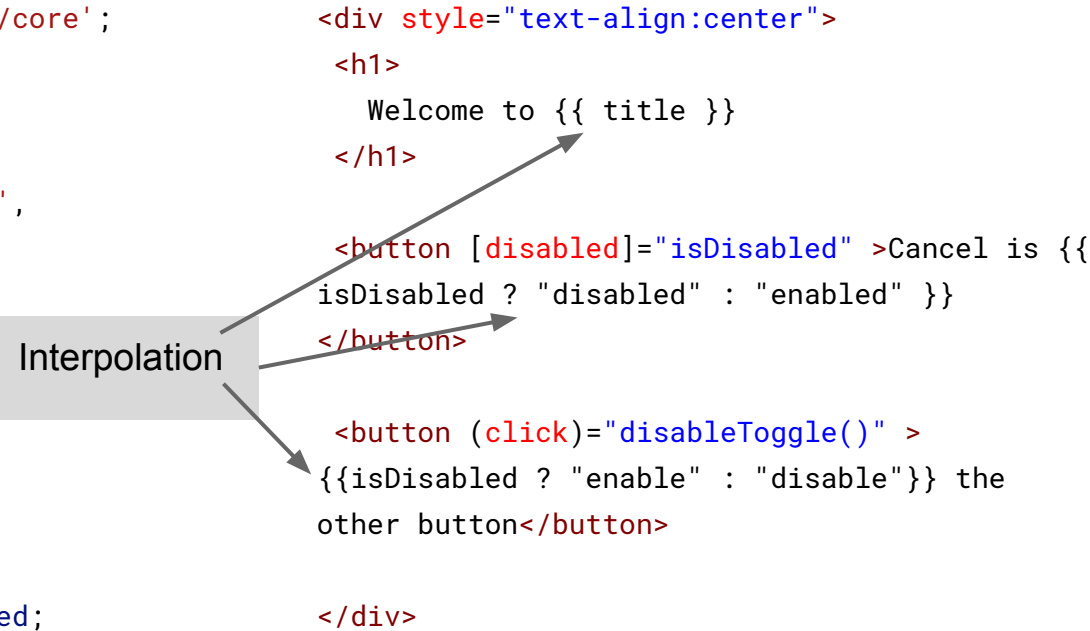
Data binding

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

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

```
export class AppComponent {  
  title = 'app';  
  isDisabled = true;  
  
  disableToggle() {  
    this.isDisabled = !this.isDisabled;  
  }  
}
```

Interpolation



```
<div style="text-align:center">  
  <h1>  
    Welcome to {{ title }}  
  </h1>  
  
  <button [disabled]="isDisabled" >Cancel is {{  
isDisabled ? "disabled" : "enabled" }}  
  </button>  
  
  <button (click)="disableToggle()" >  
    {{isDisabled ? "enable" : "disable"}} the  
    other button</button>  
  
</div>
```

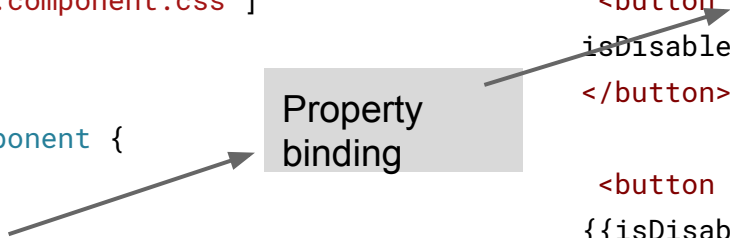
Data binding

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

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

```
export class AppComponent {  
  title = 'app';  
  isDisabled = true;  
  
  disableToggle() {  
    this.isDisabled = !this.isDisabled;  
  }  
}
```

Property
binding



```
<div style="text-align:center">
```

```
  <h1>
```

```
    Welcome to {{ title }}
```

```
  </h1>
```

```
  <button [disabled]="isDisabled" >Cancel is {{  
isDisabled ? "disabled" : "enabled" }}  
</button>
```

```
  <button (click)="disableToggle()" >  
    {{isDisabled ? "enable" : "disable"}} the  
    other button</button>
```

```
</div>
```

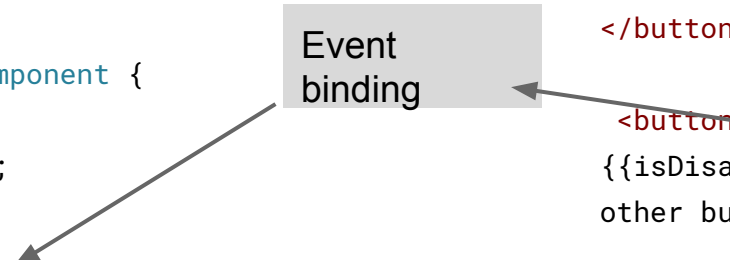

Data binding

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

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

```
export class AppComponent {  
  title = 'app';  
  isDisabled = true;  
  
  disableToggle() {  
    this.isDisabled = !this.isDisabled;  
  }  
}
```

Event
binding



```
<div style="text-align:center">
```

```
<h1>
```

```
  Welcome to {{ title }}
```

```
</h1>
```

```
  <button [disabled]="isDisabled" >Cancel is {{  
isDisabled ? "disabled" : "enabled" }}  
</button>
```

```
  <button (click)="disableToggle()" >  
    {{isDisabled ? "Enable" : "Disable"}} the  
    other button</button>
```

```
</div>
```

Restrizioni

1. Le espressioni, utilizzate nell'interpolation e nel property binding devono essere **Idempotenti**: devono poter essere valutate più volte senza cambiare lo stato dell'applicazione

Ex: `{{ counter = counter + 1 }}` non è idempotente e genera pertanto un errore

Restrizioni

2. Non è possibile accedere ad oggetti che si trovano al di fuori del contesto definito dal componente del template. Ciascun metodo o proprietà è soltanto quella definita all'interno del componente

Ex: `{{ Math.floor(0.1) }}` non è un'espressione valida perchè `Math` fa parte dell'oggetto globale e non è in generale una property del component

Property binding o interpolation?

Nei casi in cui la proprietà mappata è una stringa è possibile utilizzare alternativamente interpolazione o property binding (il risultato è lo stesso)

`<p>` is the *interpolated* image.
`<p>` is the *property bound* image.

Attenzione: il binding si riferisce ad una proprietà del DOM e non ad un attributo HTML (anche se spesso il mapping è 1:1. Possiamo ad esempio scrivere:

```
<span [innerHTML]="title"></span>
```

Pipes

Sono classi speciali che permettono di trasformare dati in input in un output desiderato. Possono essere usati nel data interpolation con la sintassi:

```
{{ expression | pipe:parameter }}
```

Esistono molteplici pipe built-in come `date`, `lowercase`, `uppercase`, `async`, etc.

Possono essere creati custom implementando la classe `PipeTransform` annotata con il decorator `@Pipe`

Pipes

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})

export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15);
}

// Stampa April 15, 1988
//Invece di Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time)
```

Pipes

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
 */
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements
PipeTransform {
  transform(value: number, exponent: string): number
  {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

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

@Component({
  selector: 'app-power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{2 |
exponentialStrength: 10}}</p>
`
})
export class PowerBoosterComponent { }
```

Directive built-in

Le direttive sono dei costrutti speciali applicabili agli elementi HTML dei template che permettono di:

- Includere contenuti selettivamente
- Selezionare tra frammenti di contenuto
- Ripetere un certo frammento per ogni elemento di un array
- etc.

***ngIf**

La direttiva ***ngIf** include un elemento e il suo contenuto soltanto se l'espressione valutata come true

```
<div *ngIf="expr"></div>
```

[ngSwitch] *ngSwitchCase

La direttiva permette di scegliere diversi frammenti da includere nel documento HTML sulla base del valore dell'espressione

```
<div [ngSwitch]="num_products()">  
  <span *ngSwitchCase="1">Only one element</span>  
  <span *ngSwitchCase="2">Only two elements</span>  
  <span *ngSwitchDefault>More than 2 elements</span>  
</div>
```

*ngFor

La direttiva permette di ripetere un certo frammento più volte, iterando su una certa collezione

```
<ul>  
  <li *ngFor="let product of get_products()">  
    {{ product }}  
  </li>  
</ul>
```

Template reference variable

E' possibile creare una variabile come riferimento ad un elemento del DOM all'interno di un certo template



```
<input #phone placeholder="phone number">
```

```
<!-- lots of other elements -->
```

```
<!-- phone refers to the input element;
```

```
pass its `value` to an event handler -->
```

```
<button (click)="callPhone(phone.value)">Call</button>
```

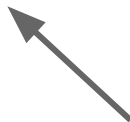
Template reference variable

Direttive speciali possono cambiare il tipo dell'oggetto puntato dalla reference variable



```
<input type="text" #username="ngModel" required name="username" [(ngModel)]="user.username"
class="form-control" id="inputUsername" placeholder="Enter username">
```

```
<div [hidden]="username.valid || username.pristine"
  class="alert alert-danger">
  Username is required
</div>
```



Valid e pristine sono proprietà di ngModel e non di HTMLFormElement.

Prima applicazione d'esempio

npm install -g @angular/cli

Angular-cli è un'utility che permette in modo semplice:

- Di generare la struttura di una nuova app
- Di generare e aggiungere components, services e moduli
- Di lanciare l'applicazione da un webserver locale, aggiornando dinamicamente il codice quando i file sorgente subiscono qualche modifica

Creare una nuova app

```
$ ng new <app_name>
```

```
$ ls <app_name>
```

README.md	karma.conf.js	
package-lock.json	protractor.conf.js	tsconfig.json
e2e	node_modules	package.json
src	tslint.json	

Aggiungere un component

\$ ng generate component comp-name

```
create src/app/comp-name/comp-name.component.css (0 bytes)
create src/app/comp-name/comp-name.component.html (28 bytes)
create src/app/comp-name/comp-name.component.spec.ts (643 bytes)
create src/app/comp-name/comp-name.component.ts (280 bytes)
update src/app/app.module.ts (486 bytes)
```

Webserver locale

\$ ng serve --open

```
** NG Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/ **
 15% building modules 45/48 modules 3 active ...form-browser/esm5/platform-browser.jswebpack: wait until
bundle finished: /
Date: 2018-03-30T20:12:14.820Z
Hash: f8743c5298a26dc080f5
Time: 6669ms
chunk {inline} inline.bundle.js (inline) 3.85 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 32 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 549 kB [initial] [rendered]
chunk {styles} styles.bundle.js (styles) 41.5 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 7.42 MB [initial] [rendered]

webpack: Compiled successfully.
```

Concetti avanzati

Dependency injection

DI è un pattern molto importante, utilizzato in molti contesti nel framework Angular per gestire le dipendenze fra componenti

Scopo: fare in modo che una classe riceva le dipendenze di cui ha bisogno da una sorgente esterna invece di istanziarle autonomamente

Esempio

```
export class Car {  
  
  public engine: Engine;  
  public tires: Tires;  
  
  constructor() {  
    this.engine = new Engine();  
    this.tires = new Tires();  
  }  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with ` +  
      `${this.engine.cylinders} cylinders  
and ${this.tires.make} tires.`;  
  }  
}
```

La classe **Car** per funzionare ha bisogno di **Engine** e **Tires**

- **Car** istanzia un oggetto **Engine** e un oggetto **Tires** nel proprio costruttore
- La classe **Car** è “fragile” perché dipende fortemente non solo dall'interfaccia delle dipendenze ma anche dalla loro implementazione

Problemi

```
export class Car {  
  
  public engine: Engine;  
  public tires: Tires;  
  
  constructor() {  
    this.engine = new Engine();  
    this.tires = new Tires();  
  }  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with ` +  
      `${this.engine.cylinders} cylinders and  
      ${this.tires.make} tires.`;  
  }  
}
```

- Cosa succede se la classe Engine viene modificata in modo che il costruttore richieda dei parametri?
- Cosa succede se in futuro viene creata una sottoclasse di Tires che vorremmo usare nella nostra classe Car?
- Cosa succede se volessimo aggiungere a Car una classe (servizio) condiviso con altre istanze di Car?

Problemi

```
export class Car {  
  
  public engine: Engine;  
  public tires: Tires;  
  
  constructor() {  
    this.engine = new Engine();  
    this.tires = new Tires();  
  }  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with ` +  
      `${this.engine.cylinders} cylinders and  
      ${this.tires.make} tires.`;  
  }  
}
```

- Testare la classe Car è complesso perché dobbiamo essere in grado di istanziare tutte le sue dipendenze
 - Le dipendenze potrebbero essere incompatibili con il nostro ambiente di test (che succede se Engine effettua una chiamata asincrona ad un server?)

Soluzione

```
export class Car {  
  
  constructor(public engine: Engine, public  
    tires: Tires) {}  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with ` +  
      `${this.engine.cylinders} cylinders  
and ${this.tires.make} tires.`;  
  }  
}
```

Nel pattern DI la definizione delle dipendenze è specificata nel costruttore.

- La classe Car si aspetta di ricevere le dipendenze di cui ha bisogno dall'esterno già pronte per l'uso
- La classe Car non istanzia più automaticamente le sue dipendenze che vengono "iniettate" dall'esterno

DI

```
// Simple car with 4 cylinders let car  
= new Car(new Engine(), new Tires());
```

```
class Engine2 {  
    constructor(public cylinders:  
number) {}  
}
```

```
// Super car with 12 cylinders  
let bigCylinders = 12;  
let car = new Car(new  
Engine2(bigCylinders), new Tires());
```

- La classe Car è ora dipendente soltanto dall'interfaccia degli oggetti che utilizza
- Il problema di istanziare gli oggetti è stato spostato **dal produttore al consumatore** dell'oggetto Car

Injector

Per far sì che anche il consumatore non sia costretto a gestire manualmente la creazione delle dipendenze, un framework di DI fornisce in genere un componente chiamato **injector** che:

- Gestisce una lista di oggetti “iniettabili” come dipendenze
- Conosce come costruire ciascun oggetto
- E' in grado di istanziare ciascun oggetto iniettabile on-demand

Servizi e Componenti

In Angular un servizio (Service) è una classe che può essere iniettata all'interno di un Component per fornire funzionalità specifiche solitamente indipendenti dalla logica di visualizzazione

Ex:

- Un component mostra la lista di utenti
- Un servizio permette di recuperare la lista da un RESTful webservice

Servizi e Componenti

Un componente solitamente non implementa le funzionalità che esulano dalla gestione della view come:

- Gestire il recupero e invio di dati ad un server
- Validare l'input degli utenti
- Logging sulla console
- etc.

Utilizza invece dei servizi che possono essere riutilizzati in più componenti

Servizi e Componenti

Un servizio è una qualsiasi classe decorata con il decorator `@Injectable`.

- Un componente può utilizzare un servizio definendolo come dipendenza all'interno del suo costruttore
- Durante il processo di bootstrap, Angular crea un Injector per iniettare i servizi all'interno dei componenti

Injector

- L'injector di Angular mantiene in memoria tutte le dipendenze già istanziate per poterle riutilizzare se più componenti necessitano dello stesso servizio (L'injector è di fatto una factory di dipendenze)

Come fa l'injector a sapere come creare le dipendenze?

- Utilizzando un provider

Provider

Per ogni dipendenza necessaria all'interno di un'applicazione, occorre registrare un provider all'interno dell'injector di modo che possa creare nuove istanze della dipendenza ogni volta che queste sono richieste da un componente.

Per i servizi, il provider è semplicemente la classe stessa. E' possibile creare provider custom per i casi particolari

Esempio

Creiamo un servizio:

\$ ng generate service logger

```
create src/app/logger.service.spec.ts (374 bytes)  
create src/app/logger.service.ts (112 bytes)
```


Esempio

```
//logger.service.ts
import { Injectable } from '@angular/core';

@Injectable()
export class LoggerService {

  constructor() { }

  public log( message: string ) {
    console.log( `${new Date()}: ${message}` );
  }

}
```

Il servizio è semplicemente una classe con l'annotazione **@Injectable**.
Un servizio può dipendere a sua volta da altri servizi

Esempio

```
import { Component } from '@angular/core';
import { LoggerService } from '../logger.service';

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

  products: {id: number, name: string}[] = [];
  private idgen = new UniqueId();

  constructor( private log: LoggerService ) {}

  add_product( p: string ) {
    this.log.log('Adding new product');
    this.products.push( {id: this.idgen.get(), name: p } );
  }

  delete_product( id: number ) {
    this.log.log( 'Deleting product with id ' + id );
    /*....*/ }
}
```

Il Componente che vuole utilizzare il servizio inserisce la sua dichiarazione nel costruttore.

E' fondamentale annotare il parametro con il tipo corretto che sarà la chiave usata dall'injector per capire che istanza iniettare

Esempio

```
import { Component } from '@angular/core';
import { LoggerService } from '../logger.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [
    {provide: LoggerService, useClass: LoggerService }
  ],
})
export class AppComponent {

  products: {id: number, name: string}[] = [];
  private idgen = new UniqueId();
  constructor( private log: LoggerService ) {}
  add_product( p: string ) {
    this.log.log('Adding new product');
    this.products.push( {id: this.idgen.get(), name: p } );
  }
  delete_product( id: number ) {
    this.log.log( 'Deleting product with id ' + id );
    /*....*/ }
}
```

Per rendere il servizio disponibile soltanto al componente, possiamo registrare il provider nella lista di providers utilizzata dal componente.

Questi sono definibili nei metadati del componente stesso

Esempio

```
import { Component } from '@angular/core';
import { LoggerService } from '../logger.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [
    {provide: LoggerService, useClass: LoggerService }
  ],
})
export class AppComponent {

  products: {id: number, name: string}[] = [];
  private idgen = new UniqueId();
  constructor( private log: LoggerService ) {}
  add_product( p: string ) {
    this.log.log('Adding new product');
    this.products.push( {id: this.idgen.get(), name: p } );
  }
  delete_product( id: number ) {
    this.log.log( 'Deleting product with id ' + id );
    /*....*/ }
}
```

Chiave

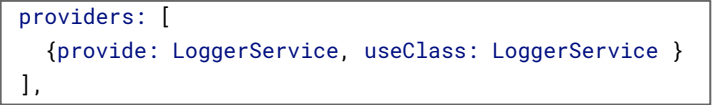
Classe da istanziare. Può essere diversa dalla chiave purché implementi la stessa interfaccia

Esempio

```
import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';
```

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

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule  
  ],  
  providers: [  
    {provide: LoggerService, useClass: LoggerService }  
  ],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```



Per rendere il servizio disponibile in tutta l'applicazione, lo stesso provider può essere registrato direttamente nella definizione del modulo root

In questo modo sarà usata **una sola istanza** di LoggerService per tutti i componenti del modulo

DI gerarchico

Esistono in realtà molteplici injectors, uno per ogni componente presente nell'applicazione.

La gerarchia degli injectors è identica alla gerarchia dei componenti

Le dipendenze vengono iniettate rispettando l'ordine gerarchico tra i componenti

Injector bubbling

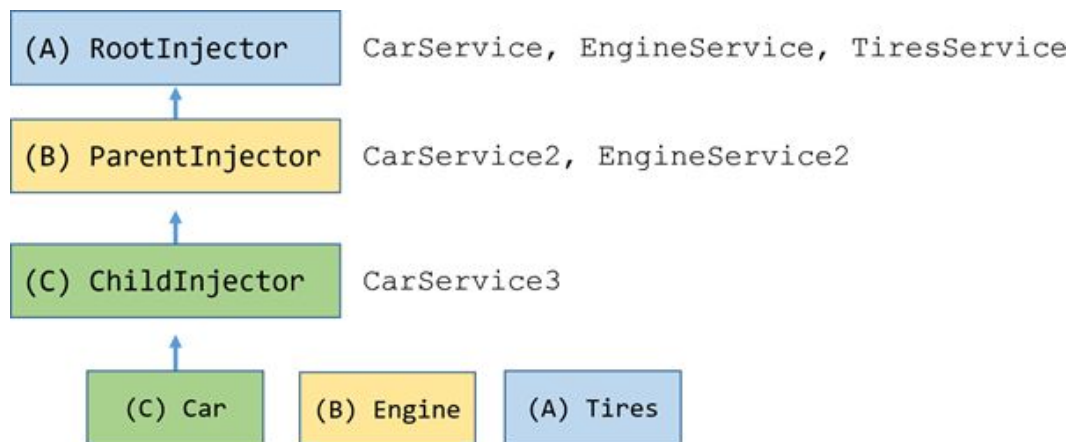
Quando un componente richiede una dipendenza, Angular cerca di risolverla usando il provider registrato sull'injector del componente stesso

Se l'injector del componente non contiene il provider, la richiesta è delegata all'injector del componente padre, risalendo la gerarchia dal basso verso l'alto

Se nemmeno l'injector del modulo contiene il provider giusto, viene generato un errore

Specialized providers

Un vantaggio della DI gerarchica è che possiamo fornire servizi con diverso grado di specializzazione in diversi punti della gerarchia



Observables

Un altro pattern molto importante utilizzato da Angular è chiamato “Observer” ed è utilizzato per passare messaggi tra due entità, chiamate rispettivamente **publisher** e **subscriber**.

Sono fondamentali per permettere la gestione asincrona dei dati risolvendo il problema in modo dichiarativo

Observables

Un Observable definisce una **funzione** che può pubblicare valori ad un certo **observer**.

A differenza delle promise, la funzione non viene eseguita fintanto che l'entità che vuole consumare quei valori non si “iscrive” a quel determinato Observable.

Il consumer riceve notifiche sui valori pubblicati fino a quando la funzione Observer ritorna oppure il consumer si disiscrive (unsubscribe) esplicitamente

```

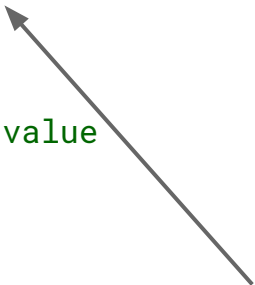
const sequenceObservable = new Observable( (observer) => {
  const seq = [1, 2, 3];
  let timeoutId;

  // Will run through an array of numbers, emitting one value
  // per second until it gets to the end of the array.
  function doSequence(arr, idx) {
    timeoutId = setTimeout(() => {
      observer.next(arr[idx]);
      if (idx === arr.length - 1) {
        observer.complete();
      } else {
        doSequence(arr, idx++);
      }
    }, 1000);
  }

  doSequence(seq, 0);

  // Unsubscribe should clear the timeout to stop execution
  return {unsubscribe() {
    clearTimeout(timeoutId);
  }};
});

```



Il costruttore di Observable prende in input la funzione che pubblica valori ad un certo observer

```

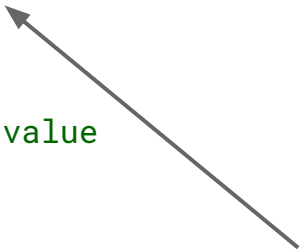
const sequenceObservable = new Observable( (observer) => {
  const seq = [1, 2, 3];
  let timeoutId;

  // Will run through an array of numbers, emitting one value
  // per second until it gets to the end of the array.
  function doSequence(arr, idx) {
    timeoutId = setTimeout(() => {
      observer.next(arr[idx]);
      if (idx === arr.length - 1) {
        observer.complete();
      } else {
        doSequence(arr, idx++);
      }
    }, 1000);
  }

  doSequence(seq, 0);

  // Unsubscribe should clear the timeout to stop execution
  return {unsubscribe() {
    clearTimeout(timeoutId);
  }};
});

```



Observer è un oggetto che contiene 3 funzioni:

- `next(v)` è l'handler che gestisce ciascun valore `v` che viene generato
- `error(e)` è l'handler che gestisce eventuali errori
- `complete()` *[opzionale]* è l'handler che viene invocato quando non ci sono più valori da pubblicare

Subscribers

Creare un Observable non comporta l'esecuzione della funzione. Per iniziare a ricevere i valori è necessario invocare il metodo `subscribe` passando come parametri 3 funzioni: `next`, `error` e `complete`:

```
sequenceObservable.subscribe(  
    (x) => { console.log("New value generated: "+x); },  
    (err) => { console.log("Error: " + err );  
    () => { console.log("No more values");  
});
```

Observables vs. promises

- Gli Observables sono dichiarativi, permettono di creare delle “ricette” che possono essere eseguite quando un componente invoca subscribe. Le promise vengono invece eseguite appena vengono create.
- Gli observable pubblicano una sequenza anche infinita di valori. Le promise forniscono un solo valore

Observables vs. promises

- Gli observer possono creare delle catene di trasformazione sui valori, che vengono eseguiti solo quando un consumer invoca subscribe:
 - `observable.map((v)=>2*v)`
- Un observable può essere cancellato mentre è in esecuzione invocando il metodo `unsubscribe()`



Reactive Extensions for JavaScript è una libreria utilizzata da Angular che utilizza gli observables per rendere più semplice lo sviluppo di codice asincrono basato su callback.

Fornisce l'implementazione della classe Observable e delle utility per creare Observables da altri tipi (come array o promise) e modificare lo stream di valori prodotti



Ex.1 Creare un observable da una sequenza di valori:

```
const nums = Observable.of(1, 2, 3);
```

Ex.2 Creare un nuovo observable che modifica al volo i valori prodotti dall'observable precedente

```
const squareValues = nums.map((val: number) => val * val);  
squareValues.subscribe(x => console.log(x));
```

```
// Logs  
// 1 4 9
```

RxJS pipe

L'operatore pipe permette di combinare più funzioni in una singola funzione che le esegue in sequenza per ogni valore generato

```
import { filter } from 'rxjs/operators/filter';
import { map } from 'rxjs/operators/map';

const squareOdd = Observable.of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );

// Subscribe to get values
squareOdd.subscribe(x => console.log(x)); // 4 16
```

RxJS operators

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

EventEmitter

E' una classe fornita da Angular che implementa un Observer aggiungendo la funzione `emit(v)`. Quando invocata, l'observer pubblica il valore `v` al consumer.

```
@Component({
  selector: 'zippy',
  template: `<div class="zippy">
    <div (click)="toggle()">Toggle</div>
    <div [hidden]="!visible">
      <ng-content></ng-content>
    </div> </div>`})
export class ZippyComponent {
  visible = true;
  public visibility_change = new EventEmitter<any>();

  toggle() {
    this.visible = !this.visible; if (this.visible) { this.visibility_change.emit(true); }
  }
}
```

HttpClient

E una delle classi più utilizzata di Angular.

Permette di effettuare chiamate HTTP asincrone e di ritornare i valori ottenuti sotto forma di observable.

```
Observable<Users[]> o = http.get<Users[]>('http://myserver.com/api/v1/users')
    .pipe(
        tap(users => this.log(`fetched users: `+JSON.stringify(users) )),
        catchError(this.handleError('getHeroes', []))
    );

o.subscribe( (users)=>{ this.users = users } );
```

HttpClient

I metodi `get` e `delete` si aspettano di default dati formattati come stringa JSON.

E' possibile specificare headers aggiuntivi, parametri della richiesta, etc.

Nota: La richiesta HTTP ritorna un observable.
Pertanto, la richiesta viene effettuata soltanto quando è invocato il metodo `subscribe()`

HttpClient

Utilizzo:

1. Iniettare il servizio HttpClient nel proprio componente o servizio

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


2. Effettuare la richiesta

```
http.get( <url>:string, <options>:{} ) : Observable< any >
```

HttpClient get options

```
options: {  
  headers?: HttpHeaders | {  
    [header: string]: string | string[];  
  };  
  observe?: HttpObserve;  
  params?: HttpParams | {  
    [param: string]: string | string[];  
  };  
  reportProgress?: boolean;  
  responseType?: 'arraybuffer' | 'blob' | 'json' |  
'text';  
  withCredentials?: boolean;  
}
```

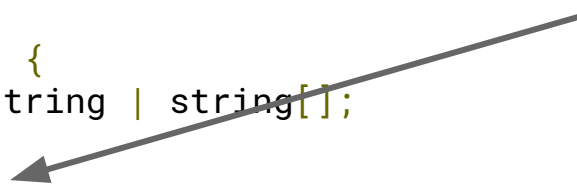
Eventuali headers
aggiuntivi come
coppie chiave: valore



HttpClient get options

```
options: {  
  headers?: HttpHeaders | {  
    [header: string]: string | string[];  
  };  
  observe?: HttpObserve;  
  params?: HttpParams | {  
    [param: string]: string | string[];  
  };  
  reportProgress?: boolean;  
  responseType?: 'arraybuffer' | 'blob' | 'json' |  
  'text';  
  withCredentials?: boolean;  
}
```

Se impostato,
l'observable ritorna la
response completa
(con headers, valore
di ritorno, etc) e non
soltanto i dati



HttpClient get options

```
options: {  
  headers?: HttpHeaders | {  
    [header: string]: string | string[];  
  };  
  observe?: HttpObserve;  
  params?: HttpParams | {  
    [param: string]: string | string[];  
  };  
  reportProgress?: boolean;  
  responseType?: 'arraybuffer' | 'blob' | 'json' |  
  'text';  
  withCredentials?: boolean;  
}
```

Parametri della
request che vengono
automaticamente
serializzati secondo il
tipo MIME
application/x-www-f
orm-urlencoded

Invio dati al server

I metodi `post()` e `put()` di `HttpClient` permettono l'invio di dati al server. La funzione supporta l'invio di dati generici (oggetti) che vengono solitamente serializzati in stringhe JSON

```
http.post( <url>:string, <body>: any, <options>:{} ) :  
Observable< any >
```

Routing

Angular fornisce un componente speciale chiamato Router che implementa una navigazione tra diverse view con un modello analogo a quello comunemente utilizzato dal browser:

- L'URL indica la view visualizzata
- Cliccando su un link la view viene sostituita con quella puntata da un link
- I pulsanti back e forward permettono di navigare avanti e indietro nella storia delle view visitate

Routing

Il modello di navigazione implementato dal router è simile a quello del browser ma il tipo di applicazione web resta di tipo **single page**:

- La modifica della view corrente non necessariamente causa una chiamata HTTP
- Può essere modificata soltanto una porzione della pagina corrente (una view), non tutta la pagina
- Gli eventi di routing possono essere intercettati e scatenare altre azioni

Impostare il routing

Il primo step è specificare qual'è l'URL di base da cui il router può comporre gli URL per tutte le sotto-view.

L'URL di base è impostato nel file index.html con il tag `<base>`:

```
<base href="/" >
```

Impostare il routing

Si genera poi un modulo e lo si aggiunge al progetto:

```
$ ng generate module app-routing --flat --module=app
```

Nome modulo



Non generare una dir
dedicata per il modulo

Importa automaticamente il
nuovo modulo nel modulo
“app” (solitamente è il nome
del root module)

Impostare il routing

Si specificano le route istanziando un oggetto Routes

<https://angular.io/api/router/Routes>

```
const routes: Routes = [  
  { path: '', redirectTo: '/login', pathMatch: 'full' },  
  { path: 'login', component: UserLoginComponent },  
  { path: 'messages', component: MessageListComponent }  
];
```

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


Impostare il routing

Si invoca il metodo `forRoot()` che ritorna un **modulo** con tutti i servizi e direttive necessarie per il suo utilizzo. Il modulo è configurato con le route appena definite ed esportato agli altri componenti dell'applicazione

```
const routes: Routes = [
  { path: '', redirectTo: '/login', pathMatch: 'full' }, { path: 'login', component:
  UserLoginComponent }, { path: 'messages', component: MessageListComponent }
];
@NgModule({
  imports: [ RouterModule.forRoot(routes) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

Impostare il routing

A questo punto è possibile inserire la direttiva `<router-outlet>` nel template di un qualsiasi component o direttamente in `index.html`. L'elemento specifica in quale posto del DOM vanno inserite le varie view gestite dal router

```
<h1>{{title}}</h1>
```

```
<router-outlet></router-outlet>
```

```
<app-messages></app-messages>
```

Navigazione

E' possibile cambiare view inserendo dei link all'interno del codice HTML, come si farebbe con le normali pagine:

```
<a routerLink="/signup">Sign-up new user</a>
```

oppure programmaticamente nel codice di un component

```
constructor( private router: Router ) { }  
change_page( ) {  
    this.router.navigate(['/page']);  
}
```

Per saperne di più

Documentazione ufficiale:

<https://angular.io/docs>

Framework API list

<https://angular.io/api>

Cheat sheet

<https://angular.io/guide/cheatsheet>