Applicazioni e framework per PWAs:
Angular Core

Ing. Luigi Brandolini

Agenda

- Introduzione
- Version History
- CLI & ambiente di sviluppo
- Moduli, Componenti e Servizi
- Direttive: strutturali e customs
- Data Bindings
- Forms
- Service
- Observables
- HttpClient
- Routing

Versioni

- **Angular 2**

- Angular 3: *never released*

- Angular 4:
  - Ahead of Time Compilation (AoT): migliora la gestione degli errori
  - Angular Routing

- Angular 5:
  - Progressive Web Application (PWAs) support

Versioni

- Angular 6
  - CLI improved
  - Angular elements
- Angular 7
  - Bug fixing
  - Small improvements

Versioni

- Angular 8
  - TypeScript 3.4
  - Ivy (compiler/runtime preview)
  - Forms improvements
  - Router: new TS syntax for lazy Loading modules
  - New `'static'` option added for `ViewChild` and `ContentChild`
  - Improvements on Service worker registration strategy

Versioni

- Angular 9
  - Ivy compiler (default)
    - Smaller bundle sizes
    - Faster testing
    - Better debugging
    - Improved CSS class and style binding
    - Improved type checking
    - Improved build errors
    - Improved build times, enabling AOT on by default
    - Improved Internationalization
  - Improvements various (ngUpdate, new components, Visual Studio Code new features, ..)

Versioni

- Angular 10
  - Libraries updates (es. TS 3.9 support)
  - Angular Material now includes a new date range picker
  - Smaller improvements/fixes
- Angular 11
  - Faster Builds
  - TSLint deprecated (ESLint recommended)
  - IE support removed for old versions (9, 10 e mobile)
  - Webpack 5 Support
  - Improved Logging and Reporting

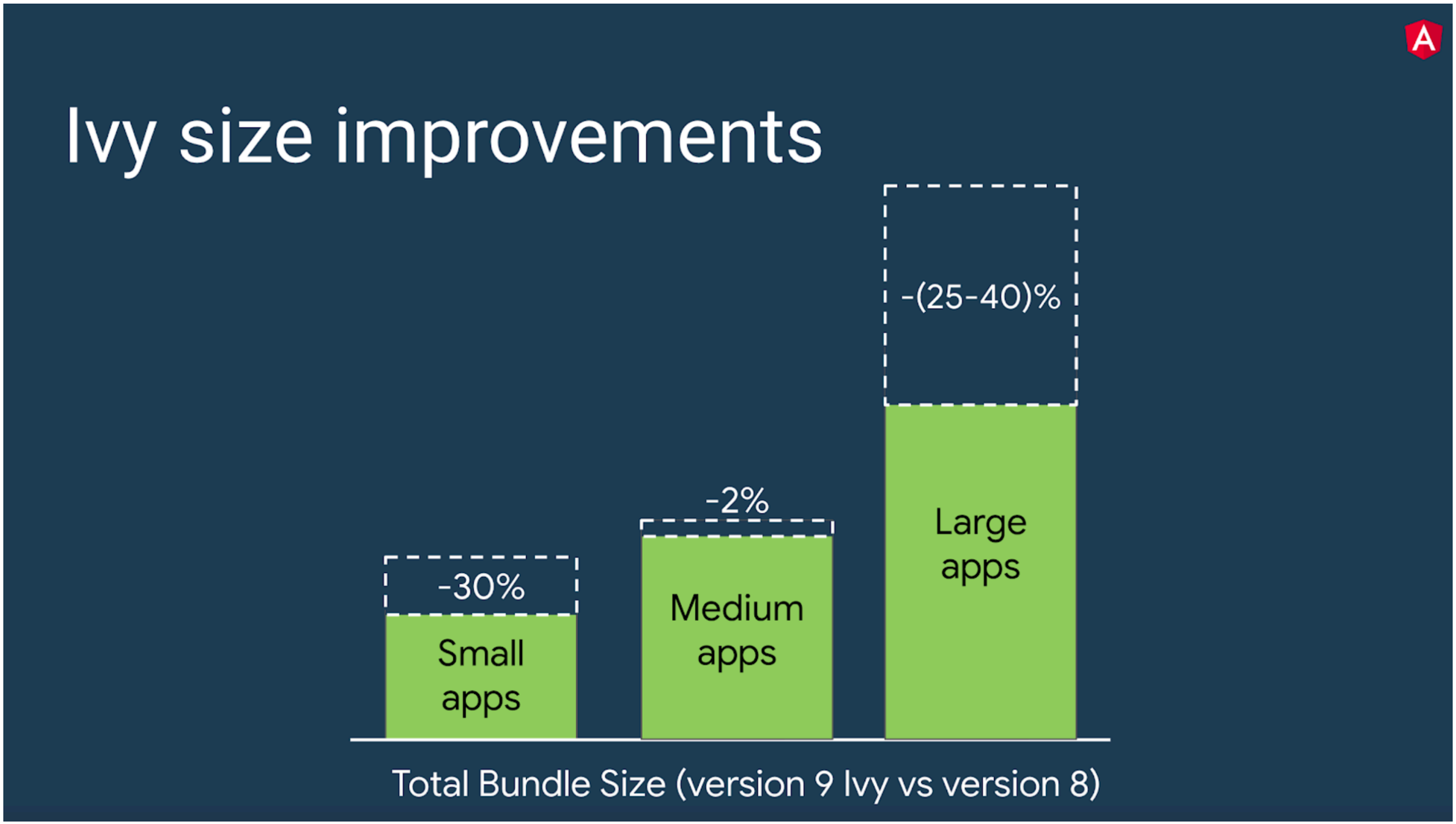
Versioni

- Angular 12
  - View Engine deprecated
  - Transition from i18n Legacy to Message IDs
  - Nullish Coalescing Operator added to Templates
  - Stylish Improvements (SASS supported in line within components)
  - Deprecating Support for IE11
  - Webpack 5 production ready supported

Versioni

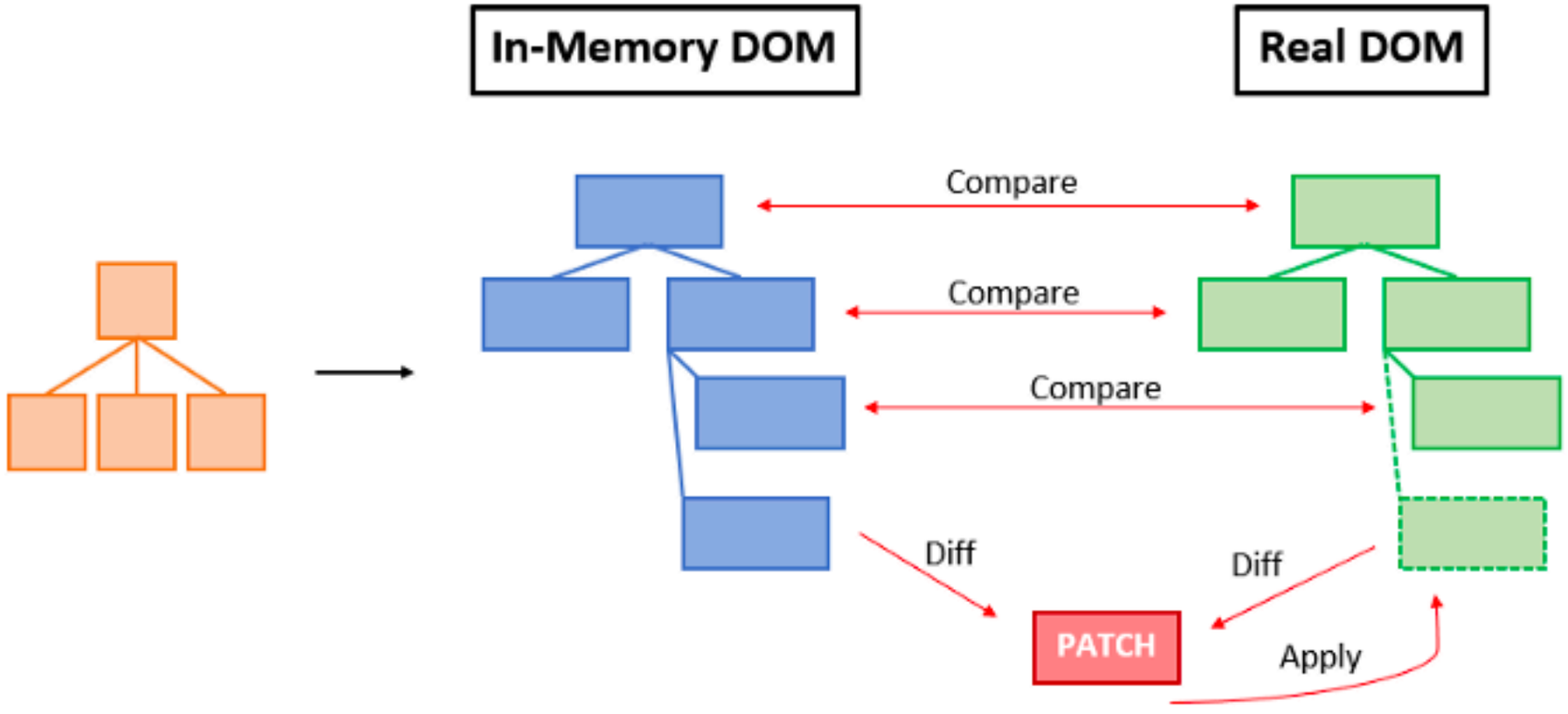
- **Angular 13**
  - View Engine support dropped
  - IE not supported anymore
  - Support to TypeScript 4.4
  - RxJS 7.4 integration
  - Router improvements
  - Angular CLI enhancements

Angular Ivy (a partire dalla versione 9)

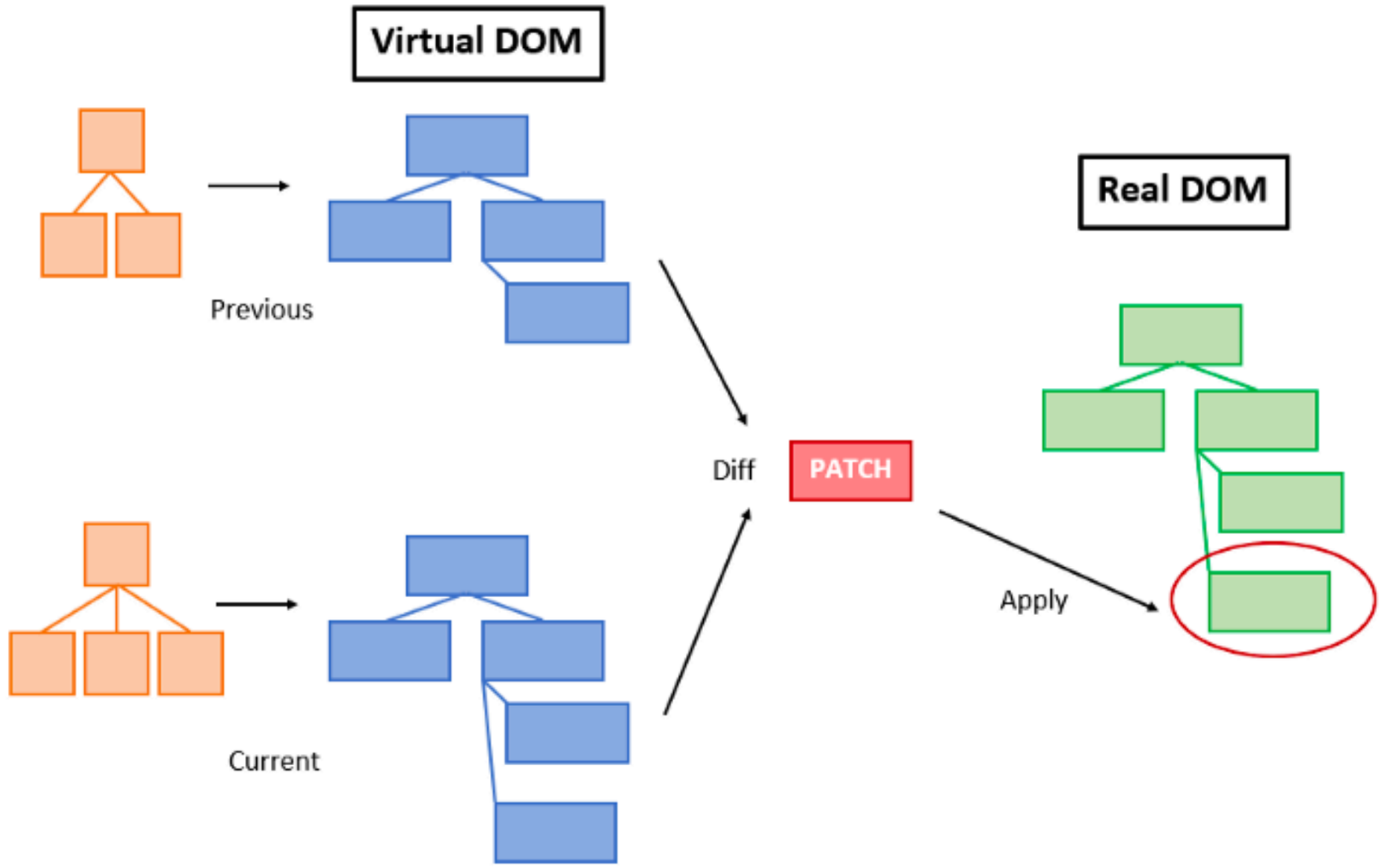


Incremental DOM

- Ogni componente viene compilato in un insieme di istruzioni
- Da qui viene creato il DOM tree e apportati i cambiamenti ad esso



Virtual DOM



Angular CLI

- Per semplificare lo sviluppo è stato introdotto un ambiente a riga di comando per creare la struttura di un'applicazione Angular 2+:

```
npm install -g @angular/cli
```

(in alcuni sistemi eseguire in modalità'sudo')

Angular CLI

- In caso di problemi futuri, per re-installare la cli di Angular:

update angular-cli to the latest version

```
$ npm remove -g angular-cli
$ npm install -g @angular/cli@latest
```

update the project dependencies

```
$ rm -rf node_modules dist
$ npm install --save-dev @angular/cli@latest
$ npm install
```

Angular CLI

PROJECT UPDATE:

Aggiornare il progetto Angular alla versione attuale della piattaforma di sviluppo:

`ng update`
(dopo aver aggiornato Angular CLI)

Angular CLI - Utilizzo

- Per creare una nuova applicazione con scaffolding di base è sufficiente digitare:

```
ng new FirstApp
```

- Per eseguire la nuova applicazione eseguendo un webserver locale, portarsi nella directory appena creata (FirstApp) e digitare:

```
ng serve [--port <number>]
```

(verificare l'indirizzo `http://localhost:4200`)

Angular CLI - Utilizzo

- Tramite CLI siamo poi in grado di generare i vari elementi che costituiranno la nostra app. Il comando che ci permette di farlo è il seguente:

```
ng generate <schematic> <name> [options]
```

https://angular.io/cli/generate

Angular CLI - Utilizzo

```
ng generate <schematic> <name>
```

- Al posto di `schematic`, potremmo andare a specificare l'elemento da generare:
  - component
  - directive
  - pipe
  - service
  - class
  - interface
  - enum
  - guard
  - interceptor
  - ..

Angular CLI - Utilizzo

Esempio:

```
ng generate component Header
```

o (in forma concisa):

```
ng g c Header
```

Angular CLI - Build

- A partire dalla versione 12 di Angular, il tool per la build sarà Web Pack
- Per effettuare una build, si usa il comando `build`:

`ng build` (o nella forma abbreviata `ng b`)

Il risultato sarà la generazione dei file da eseguire, a partire da qualche server (es.: `http-server` installabile a parte), nella directory del progetto `dist/<project_name>`

- Di default verrà generata una build già ottimizzata per la produzione
- E' anche possibile creare più profili diversi per la `build` mediante opzione `--configuration`

`(default: ng build --configuration production)`

Editor

- **Visual Studio Code**
- Atom
  - Plugin:

    `npm install atom-typescript`

    @See: `https://atom.io/packages/atom-typescript`
- Sublime Text
- WebStorm IDE

`https://www.jetbrains.com`

JIT/AOT

- Angular offre due meccanismi di compilazione dell'applicazione:
  - JIT (default fino ad Angular 8): processo di compilazione dell'applicazione delle librerie effettuato *on-the-fly*, ovvero a runtime
  - AOT (default a partire da Angular 9): processo di compilazione dell'applicazione delle librerie effettuato a *build time*

- Eseguendo i comandi `ng build` (solo build) o `ng serve` (build ed esecuzione in locale) tramite CLI, il tipo di compilazione (JIT/AOT) dipende dal valore della proprietà AOT presente nella configurazione della build dell'angular.json
- Per default, aot è impostato a true nelle nuove applicazioni

JIT/AOT

JIT - Compila il TypeScript in modalità 'Just In Time'

- Compilato nel browser
- Ogni file viene compilato separatamente
- Se il codice viene cambiato, non c'è bisogno di reload
- Adatto per lo sviluppo in locale

AOT - Compila il TypeScript durante la fase di build

- Viene compilato sulla stessa macchina (più veloce)
- Tutto il codice viene compilato insieme, inlining HTML/CSS negli scripts
- Non c'è bisogno di includere anche il compilatore (richiede la metà della dimensione dell'Angular app).
- Più sicuro perché il codice sorgente non viene esposto
- Adatto per le build di produzione

Project Build

- Compilare il progetto Angular per il rilascio:

```
ng build [options] (o ng b [options])
```

- Per maggiori dettagli:

```
https://angular.io/cli/build
```

Browsers support

BROWSER	SUPPORTED VERSIONS
Chrome	latest
Firefox	latest and extended support release (ESR)
Edge	2 most recent major versions
Safari	2 most recent major versions
iOS	2 most recent major versions
Android	2 most recent major versions

- A partire da Angular 12, il supporto ad Internet Explorer 11 è **deprecato** e da Angular 13, **rimosso**
- Per maggiori informazioni: https://angular.io/guide/deprecations#internet-explorer-11

Module

- Un modulo è un contenitore di funzionalità che consente di organizzare il codice di un'applicazione

- Permette di dichiarare componenti, direttive e pipe, esportandoli, all'occorrenza

- Consente anche l'importazione di altri moduli

- Un modulo è una classe TS contenente il decoratore `@NgModule({..})`

- Vengono caricati in maniera *eager* (default) o *lazy* (questo nel caso del routing)

@NgModule

- Il decoratore `@NgModule({..})` è una funzione che prende in input un solo oggetto contenente metadati, le cui proprietà descrivono il modulo

- Le proprietà principali, sono:

  - `declarations`: components, directives, pipes che appartengono all' `NgModule` corrente

  - `exports`: il sottoinsieme di dichiarazioni che dovrebbero essere visibili ed utilizzabili nei template della componente degli altri `NgModules`

  - `imports`: Altri moduli le cui classi esportate, sono richieste dalle componenti dichiarate nell' `NgModule` corrente

  - `providers`: Generatori di `services` accessibili in qualunque punto dell'applicazione (possono essere specificati anche sul component level)

  - `bootstrap`: La view applicativa principale, chiamata root component, che ospita tutte le altre app views (solo l'NgModule **root** dovrebbe impostarla)

Module

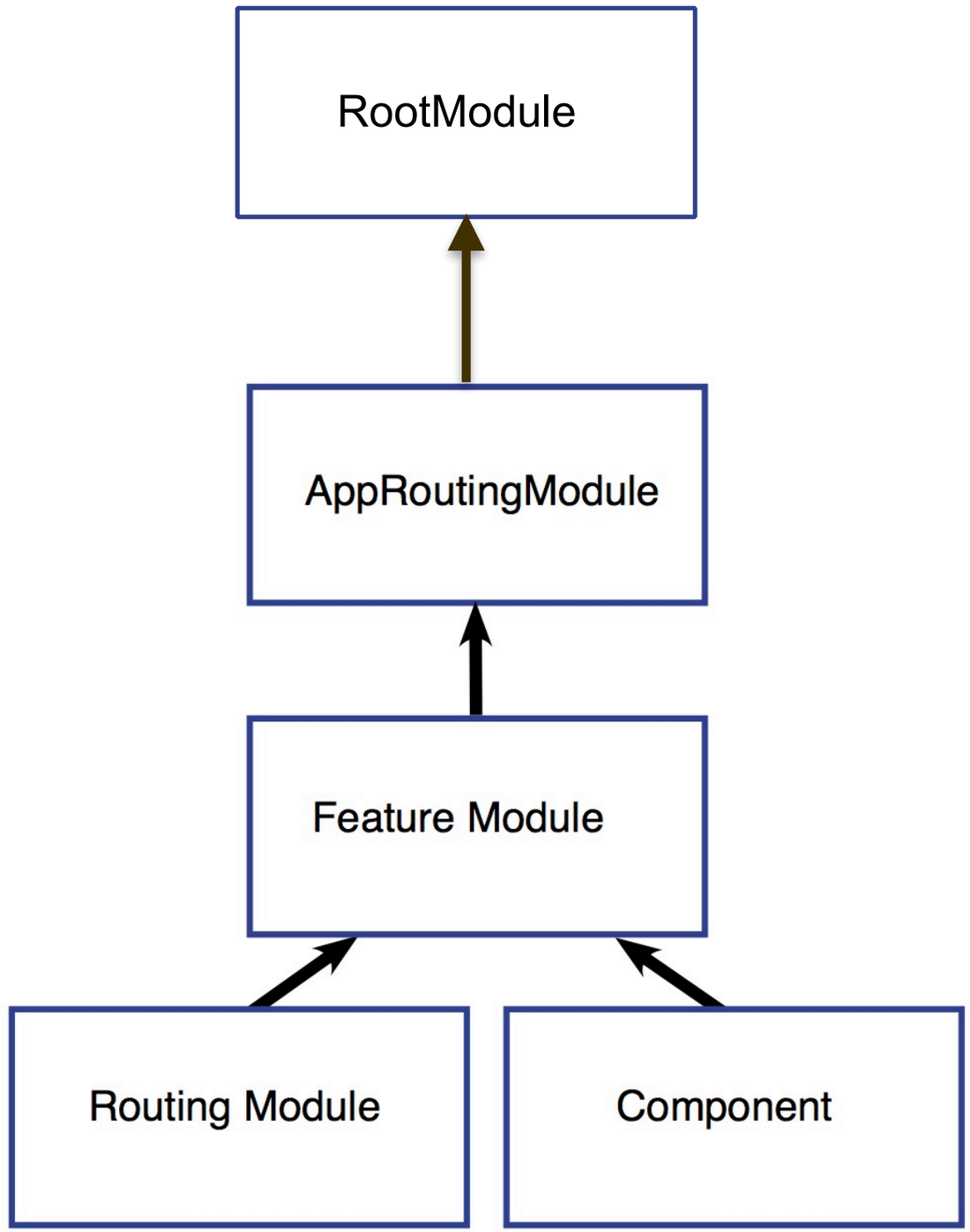
- Ogni Angular app ha almeno una classe (*root module*), chiamata convenzionalmente `AppModule` e definita nel file `app.module.ts`.

- L'esecuzione dell'applicazione avviene tramite il bootstrap dell' `NgModule` root

- L'`NgModule` viene considerato il root di un'app perché può includere `NgModule` figli in una gerarchia con profondità arbitraria

- Al crescere dell'applicazione, dovrebbero sempre essere aggiunti nuovi moduli da integrare al suo interno (*feature module* come *organizational best practice*)

Lazy Loading Feature Module



Lazy Loading Feature Module

- Il primo step consiste nella generazione di un nuovo "feature" module, tramite il supporto della CLI:

```
ng generate module items --route items --module app.module
```

- Questo step andrà ripetuto per ogni ulteriore "feature" module, eventualmente richiesto

Lazy Loading Feature Module

- Il comando precedente aggiungerà una nuova rotta nell' `AppRoutingModule`, per il feature module appena generato, riportando l'attributo **`loadChildren`**, al posto di **`component:`**

```
const routes: Routes = [
  {
      path: 'items',
      loadChildren: () => import('./items/items.module').then(m => m.ItemsModule)
  }
];
```

- Nel modulo di **routing lazy-loaded**, andrà aggiunta una rotta per ciascuna pagina/componente:

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

Component

- Alla base di Angular c'è il concetto di Component che racchiude (e sostituisce) elementi noti nella prima versione del framework come direttive, controller e scope

- Avrà inoltre il controllo di una parte dell'interfaccia grafica modellando una parte di essa, ovvero una view

- Un'applicazione può essere considerata come una raccolta di componenti in interazione tra loro

Component

- Un'applicazione continuerà ad avere un *root component* rappresentante il punto d'ingresso dell'applicazione stessa
- Tale root component per convenzione viene chiamato AppComponent

Component

- Per definire una componente è sufficiente creare una classe JavaScript con il decoratore `@Component`
- Sempre per convenzione, i nomi dei file che contengono la definizione di componenti hanno la struttura:

```
nomeComponente.component
```

Component

- Il decoratore `@Component` può contenere meta-informazioni
- Quelle di base, sono:
  - `selector:` l'elemento del markup a cui è agganciato il componente
  - `templateUrl:` il file HTML che descrive il markup del componente
  - `styleUrls:` l'elenco dei file CSS da applicare al markup
  - `encapsulation:` come dev'essere incapsulata la componente (separazione del suo stile CSS dal resto del DOM applicativo)
- Ad una componente potrebbero corrispondere più file esterni (`templateUrl e styleUrls`)

Encapsulation

- Le strategie di incapsulamento di una componente, al momento, sono le seguenti:
  - `Emulated` (default)
  - `None`
  - `ShadowDom`
- Se la policy è impostata su `ViewEncapsulation.Emulated` e il componente non ha `styles` o `styleUrls` specificati, viene modificata in `ViewEncapsulation.None`

Component

- Per definire una componente tramite CLI, eseguire il comando:

```
ng generate component <nome-componente>
```

Una volta terminata la sua esecuzione, saranno generati automaticamente i file relativi al markup HTML, CSS e TypeScript

Component

- Per ciascuna componente, viene generato un set di risorse:
  - `myComponent.ts`: file TypeScript con la definizione della componente
  - `myComponent.html`: template HTML associato alla componente
  - `myComponent.css`: foglio di stile opzionale da associare al template
  - `myComponent.spec.ts`: file per lo unit testing

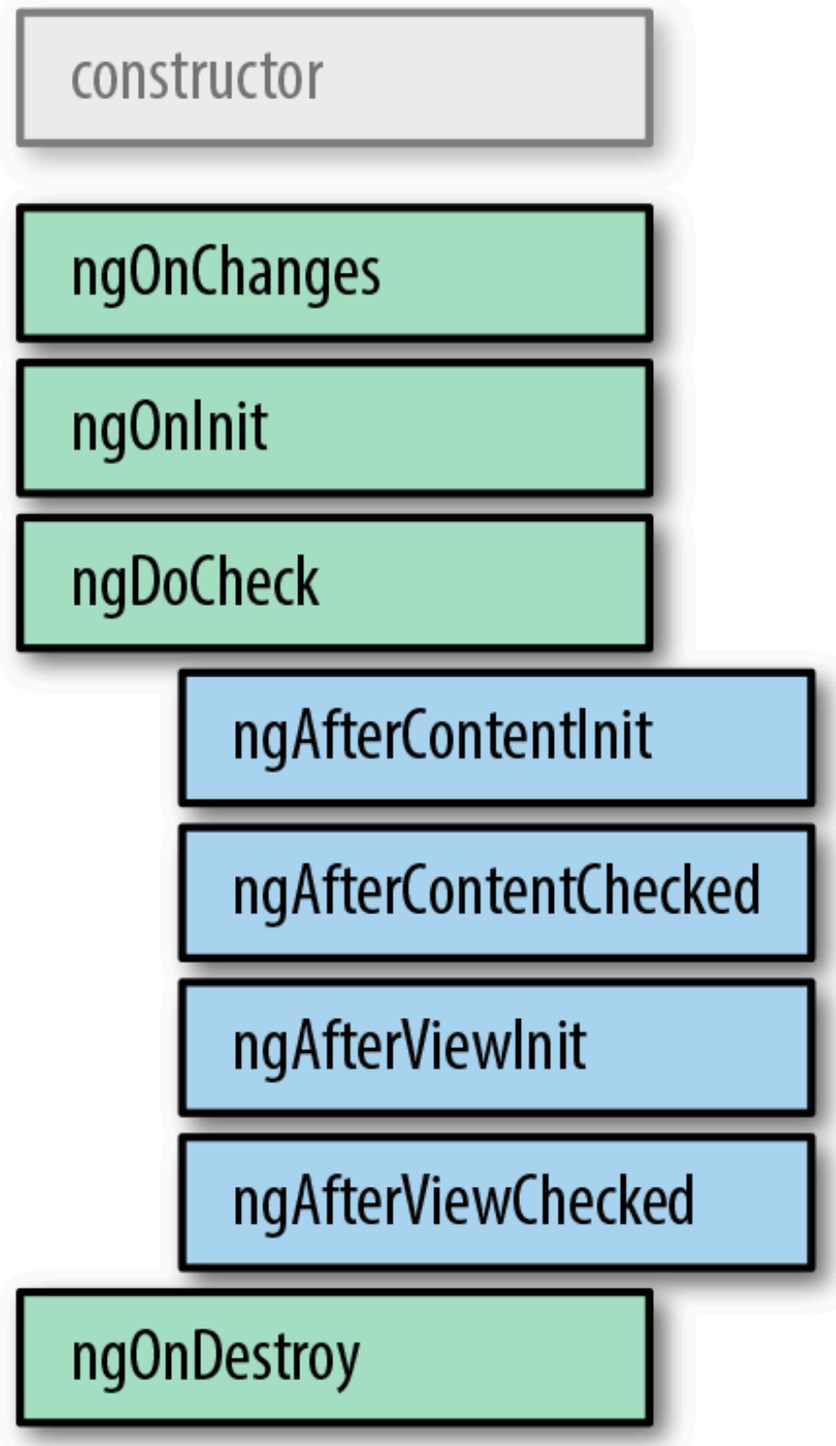
Component - Lifecycle

- Ogni componente ha un ciclo di vita strutturato in più fasi

- È importante sapere quali sono le varie fasi, per poter definire un controllo a grana più sottile

- Per intercettare le singole fasi è possibile definire degli hook (nome funzione)

Component - Lifecycle



Component - Lifecycle

- `constructor`: inizializzazione
- `ngOnChanges`: gli input type modificano il loro valore
- `ngOnInit`: termine dell'inizializzazione
- `ngDoCheck`: per implementare logiche custom di detection

NB.: sulla stessa componente non si possono utilizzare contemporaneamente `ngOnChanges e ngDoCheck`

Component - Lifecycle

- `ngOnDestroy`: invocato poco prima della distruzione della componente (da usare per l'unsubscribe degli Observables e il detaching degli event handlers, per evitare memory leaks)

Directives

- Le direttive built-in di Angular, si dividono in:

  - Structural directives: `*ngIf, *ngFor, *ngSwitch`
  - Attribute directives: `ngClass, ngStyle`

*ngIf

- Aggiunge/rimuove al DOM sub-tree quando un'espressione risulta essere verificata

Esempio:

```
<div *ngIf="error" class="alert alert-danger">{{error.message}}</div>
```

- La direttiva strutturale sarà trattata come:

```
<ng-template [ngIf]="error">
  <div class="alert alert-danger">{{error.message}}</div>
</ng-template>
```

che poi sarà trasformato in:

```
<div _ngcontent-c0>Si è verificato un errore.</div>
```

<ng-template>

- Questo tag, finalizzato alla creazione di *template fragment*, supporta meccanismi specifici come i **template variables**

- Può essere utilizzato assieme alla direttiva strutturale `*ngIf`, per esempio per modellare logiche di tipo `if/else`

Esempio:

```
<div *ngIf="feedback && feedback.success; else displayError" class="alert alert-success">
    {{ feedback.message }}
</div>

<ng-template #displayError>
    <div class="alert alert-danger">
        {{ feedback.message }}
    </div>
</ng-template>
```

*ngFor

- Aggiunge/rimuove al DOM sub-tree fino a quando un'espressione risulta essere verificata

Esempio:

```
<div *ngFor="let p of persons" class="card">{{p.fullname}}</div>
```

- La direttiva strutturale sarà trattata come:

```
<ng-template ngFor let-p [ngForOf]="persons">
  <div class="text">{{p.fullname}}</div>
</ng-template>
```

che poi sarà trasformato in:

```
<div _ngcontent-c1>Mario Rossi</div>
```

*ngSwitch

- Aggiunge/rimuove al DOM sub-tree quando l'espressione annidata fa match con la corrispondente switch expression:

```
<ul *ngFor="let person of persons"
    [ngSwitch]="person.country">
  <li *ngSwitchCase="'UK'"
      class="text-success">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'USA'"
      class="text-primary">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'HK'"
      class="text-danger">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchDefault
      class="text-warning">
    {{ person.name }} ({{ person.country }})
  </li>
</ul>
```

ngClass

- Aggiunge dinamicamente class ad un elemento HTML in base quando una determinata condizione risulta essere verificata:

```
<div *ngFor="let celeb of singers">
  <p [ngClass]="{
    'text-success':celeb.country === 'USA',
    'text-secondary':celeb.country === 'Canada',
    'text-danger':celeb.country === 'Puorto Rico',
    'text-info':celeb.country === 'India'
   }">{{ celeb.artist }} ({{ celeb.country }})
  </p>
</div>
```

ngStyle

- Aggiorna lo stile di un elemento HTML

```
<div *ngFor="let p of persons">
 <p [ngStyle]="{'background-color':p.age <= 40 ? 'green' : 'red' }">
    {{ p.name }} {{ p.surname }}
 </p>
</div>
```

View - Template Syntax

- Angular processa le espressioni per calcolare un risultato che sarà poi convertito in stringa

- Esempi di interpolation:

```
<p>The sum of 1 + 1 is {{1 + 1}}</p>

<p>{{title}}</p>

<div><img src="{{itemImageUrl}}"></div>
```

Nullish Coalescing

- A partire da Angular 12, è disponibile, nei template, una nuova sintassi che semplifica le espressioni condizionali complesse

- Esempio:

```
<div>{{age !== null && age !== undefined ?

age : calculateAge() }}</div>
```

diventa

```
<div>{{ age ?? calculateAge() }}</div>
```

Binding Syntax e Data Bindings

- Angular offre un'altra possibilità per collegare dati al modello:
  - **Event binding**

    ( ) = One-Way data binding, dalla view al modello

  - **Property binding**

    [ ] = One-Way data binding, dal modello alla view

  - **Event binding + Property binding**

    [( )] = Two-way data binding

View - Template Syntax

- Esempi:

Event binding

```
<button (click)="greet()"> Greet </button>
```

Property binding

```
<img [src]="itemImageUrl">
```

Event binding + Property binding

```
<input type="text" [(ngModel)]="user.name">
```

User Inputs events

- Per rispondere ai vari DOM events, si può far ricorso agli Event Bindings:
  - `<button (click)="onClickMe()">Click me!</button>`
  - `<input (keyup)="onKey($event)">`

```
onKey(event: any) { // without type info
  this.values += event.target.value + ' | ';
}
```

  - `<input #box (keyup.enter)="update(box.value)" (blur)="update(box.value)">`

Input & Output Properties

- Regola base: **tutte le componenti sono direttive**

- Le proprietà target di una componente possono essere impostate come `@Input()` o `@Output()`

- In base a questo principio abbiamo due meccanismi:

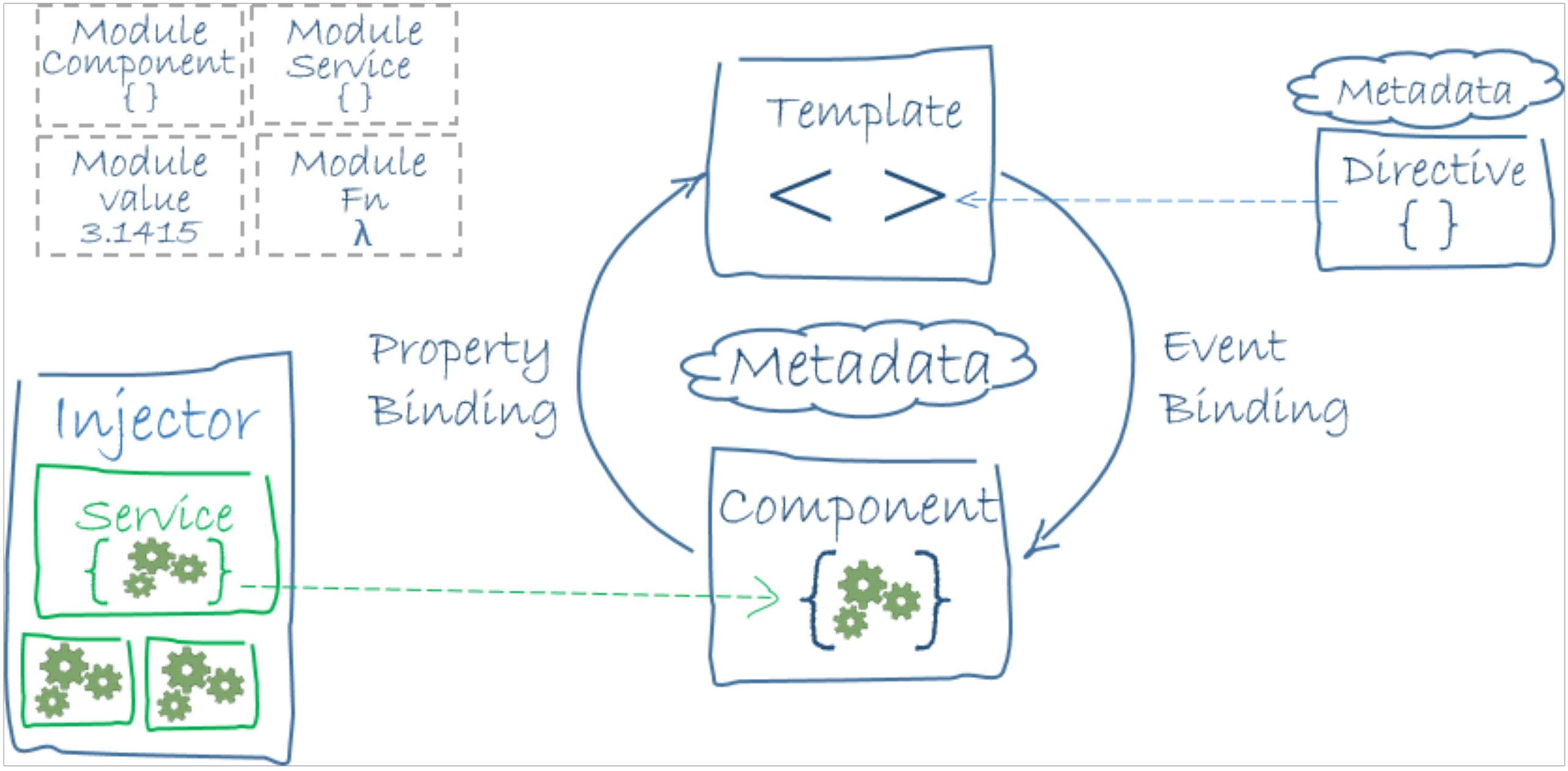
  - **`@Input()` – Property Binding:**

    consente di fare l'injection di una property dalla componente parent, alla componente child

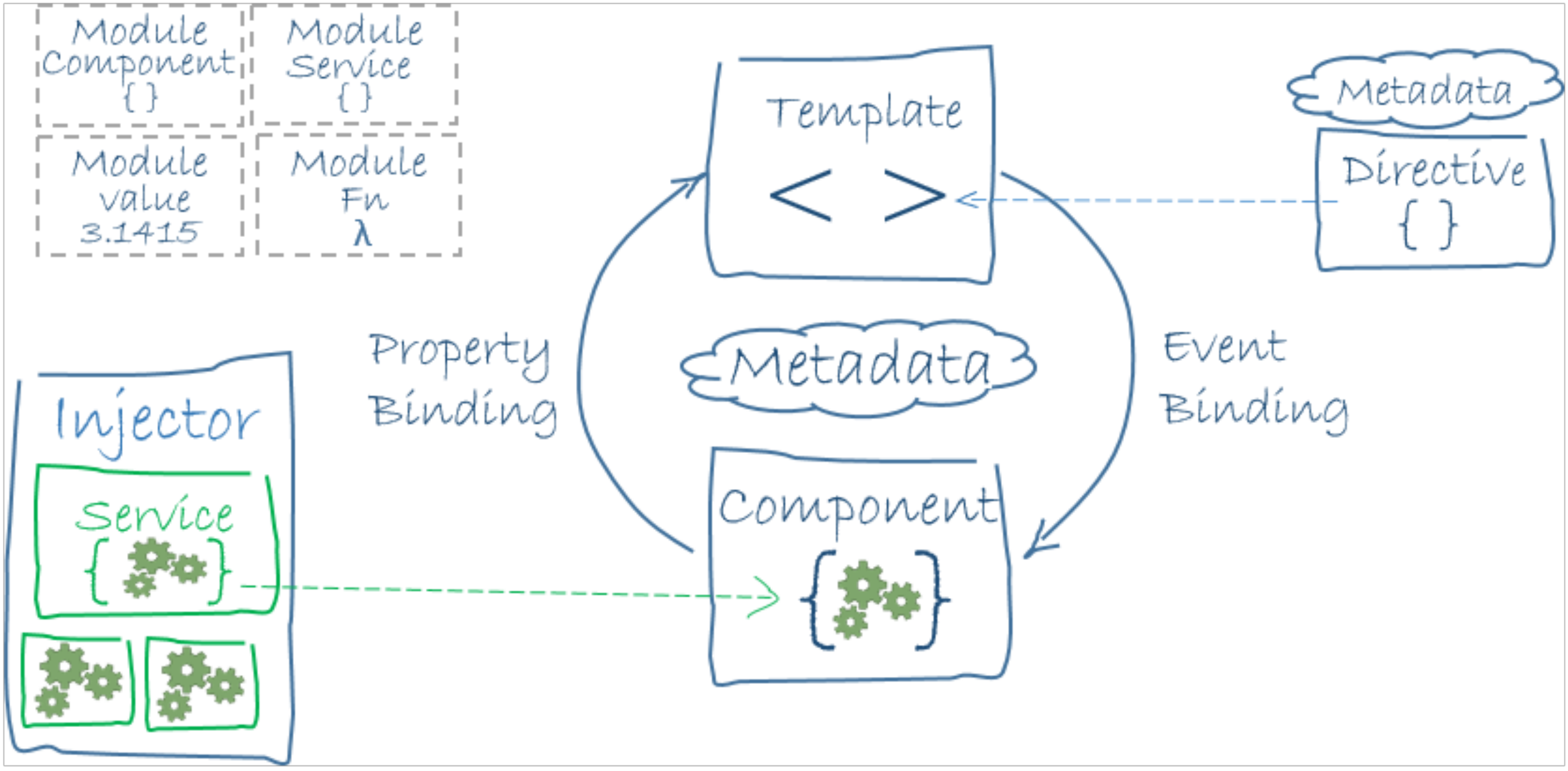
  - **`@Output()` – Event Binding:**

    consente di generare un evento dalla componente child, alla componente parent diretta

Bulding Blocks of Application



Bulding Blocks of Application



Forms

- Un Angular form:
  - tiene traccia dei cambiamenti del modello
  - effettua la validazione degli input e mostra a video gli errori

Angular Forms

- Ogni Form è composto dalle seguenti parti:
  - Grafica (Bootstrap components)
  - Modello dati
  - Regole di validità

- Il modello dati e le regole di validità possono essere implementati in due modi diversi:
  - Model Driven (*Reactive Forms*)
  - Template Driven

Validation

- Indipendentemente dalla tipologia di form, Angular offre alcuni validatori di default (`Validators` module in `'@angular/forms'`):
  - Required
  - MinLength
  - MaxLength
  - E-mail (!)
  - Pattern
- Oltre ai validatori predefiniti è ovviamente possibile definirne altri **custom**

Reactive Forms

- Richiedono l'import del modulo `ReactiveFormsModule in '@angular/form/'`
- Le regole di definizione e validazione dei form in questione, vengono espresse nella Component class attraverso l'utilizzo di funzioni
- Quando il modello cambia, Angular invoca automaticamente le funzioni definite da noi nella classe del Component
- Funzionano in maniera **sincrona**: la parte UI è sempre sincronizzata con il modello dati
- Sono costruiti sopra agli `Observables` stream (da qui l'accesso **sincrono** ai valori)

Reactive Forms

- Nel Template HTML, specificare:
  - l'attributo `[formGroup]="<formName>"` (per il tag `<form>`)
  - l'attributo `formControlName="<propertyName>"` (per il tag `<input>`)

Reactive Forms

- Le componenti core dei Reactive Forms, sono:
  - `AbstractControl` (raccoglie le proprietà astratte)
  - `FormBuilder` (inizializza il form)
  - `FormGroup` (raggruppa i form control)
  - `FormControl` (input box o selector)
  - `FormArray` (raggruppa i form group)

Reactive Forms

- Ispezionare un FormControl dal template:

```
{{ reactiveFormName.get('property').status }}
```

Es.:

```
{{ personForm.get('email').status }}
```

Reactive Forms

Property	Description
`myControl.value`	the value of a `FormControl`.
`myControl.status`	the validity of a `FormControl`. Possible values: `VALID`, `INVALID`, `PENDING`, or `DISABLED`.
`myControl.pristine`	`true` if the user has *not* changed the value in the UI. Its opposite is `myControl.dirty`.
`myControl.untouched`	`true` if the control user has not yet entered the HTML control and triggered its blur event. Its opposite is `myControl.touched`.

Template Driven Forms (TDF)

- Le regole di validazione vengono espresse mediante attributi definiti nel template HTML

- Le regole di validazione possono essere *bult-in* (`required`, `minlength`,..) o *custom* (mediante custom directives)

- Supportano una speciale sintassi chiamata Template Variables

- Richiedono l'import del modulo: `'@angular/form/FormsModule'`

- Funzionano in maniera asincrona

Template Driven Forms (TDF)

- Passi da seguire per creare un form
  - Importare `FormsModule` in `@angular/forms` nel modulo in cui utilizzeremo le capabilities
  - Creare un nuovo form HTML (eventualmente con class di Bootstrap)
  - Creare un oggetto model nella componente che utilizzerà il form
    - Collegare le data properties del model ad ogni form input mediante la sintassi ngModel (two-way data binding opzionale)
    - Aggiungere il **name** attribute a ciascun form input
    - Aggiungere al form html #<formId>="ngForm"
    - Aggiungere al form control html #<controlName>="ngModel"
    - Aggiungere CSS custom per offrire feedback visuale
    - Mostrare/nascondere messaggi di errore dopo validazione
    - Gestire il submit del form tramite ngSubmit

Form Control CSS Status

- Un Angular form offre meccanismi di two-way data binding, tiene traccia dei cambiamenti del modello, effettua la validazione degli input e mostra a video gli errori

- Come prima cosa, bisogna aggiungere al modulo dell'applicazione, l'oggetto FormsModule

Form Control CSS Status

`#<controlName>.className`

Visitato/Non visitato	ng-touched	ng-untouched
Modificato/Non modificato	ng-dirty	ng-pristine
Valido/Non valido	ng-valid	ng-invalid

Form Control CSS Status

```
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}

.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```

(Valido anche nel caso di Reactive Forms)

Custom Validation

- Le regole custom di validazione devono essere espresse all'interno di una funzione factory

- Nel caso dei Template Driven Form, sarà necessario creare una direttiva custom per collegare la regola al form control

- Nel caso dei Reactive Forms, le funzioni di validazione saranno invocate direttamente nel file TS della componente

Custom Directive

- Con una *custom directive* si può estendere la normale semantica di tag predefiniti HTML, personalizzandone l'aspetto

- Il comando per generare una direttiva custom è il seguente:

```
ng generate directive directives/<CustomDirectiveName>
```

Il risultato consisterà nella generazione di una nuova classe con decorator:

```
@Directive({
  selector: '[appCustomDirectiveName]'
})
```

- Tramite l'aggiunta di `ElementRef` nel `constructor()` è possibile effettuare l'injection di un reference all'host DOM element

- E' anche possibile utilizzare il decorator `@HostListener` per effettuare il subscribe ad eventi dell'elemento DOM che contiene la direttiva

Esempio

```
@Directive({
  selector: '[appHighlight]',
})
export class HighlightDirective {
  constructor(private el: ElementRef) {
  }

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight('');
  }

  highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

Custom Directive

- Una custom directive può essere applicata in questo modo:

```
<p appHighlight>Highlight me!</p>
```

Custom Directive e TDF

- Nel caso dei Template Driven Form è possibile usare le custom directive sul template, come wrapper delle funzioni di validazione

- Sarà necessario aggiungere, tra i metadati del decorator `@Directive`, la property **`provider`**, per far sì che Angular possa registrare la direttiva tra i validatori built-in (**`NG_VALIDATORS`**)

- La classe rappresentante la custom directive dovrà implementare l'interfaccia `Validator`

Esempio

```
import { Directive } from '@angular/core';
import { AbstractControl, NG_VALIDATORS, ValidationErrors, Validator } from
'@angular/forms';
import { validateCodFisc } from '../validators/CustomValidators';

@Directive({
  selector: '[codfisc-validator]',
  providers: [{
    provide: NG_VALIDATORS,
    useExisting: CodFiscValidatorDirective,
    multi: true
  }]
})
export class CodFiscValidatorDirective implements Validator {

  constructor() { }

  validate(control: AbstractControl): ValidationErrors {
    return validateCodFisc(control);
  }
}
```

Service

- È una component annotata con lo specifico decorator `@Injectable(providedIn: 'root') //default`

- Il comando per generare un service è il seguente:

```
ng generate service services/<ServiceName>
```

- Il decorator è in questione è stato reso disponibile a partire da Angular 6 ed indica che l'oggetto a cui è associato può essere *iniettato* all'interno di un altro oggetto (componente o servizio)

- Nelle altri componenti che ne faranno uso, va impostata la dipendenza come argomento del costruttore (***D**ependency **I**njection*)

- Ogni service viene istaziato **una sola volta**, a differenza delle Componenti vere e proprie con un meccanismo di *lazy-init*

Injectors

- In Angular, esistono due categorie di *injectors*:
  - **`ModuleInjector`**: viene configurato mediante: `@NgModule()` o `@Injectable()`
  - **`ElementInjector`**: viene creato implicitamente per ogni elemento del DOM. Per default è vuoto, a meno che non venga configurato nella property *providers* dei decorators `@Directive()` o `@Component()`

ModuleInjector

- The **ModuleInjector** può essere configurato in due modi:
  - Mediante la proprietà `providedIn di @Injectable()` che fa riferimento a `@NgModule() o a` **root**
  - Attraverso la property *`providers`* array di `@NgModule()`
- La configurazione di `@NgModule()` dell'AppModule sovrascrive `@Injectable()`, nel caso siano entrambe presenti

Service

- Marca una classe come idonea ad essere iniettata come dipendenza

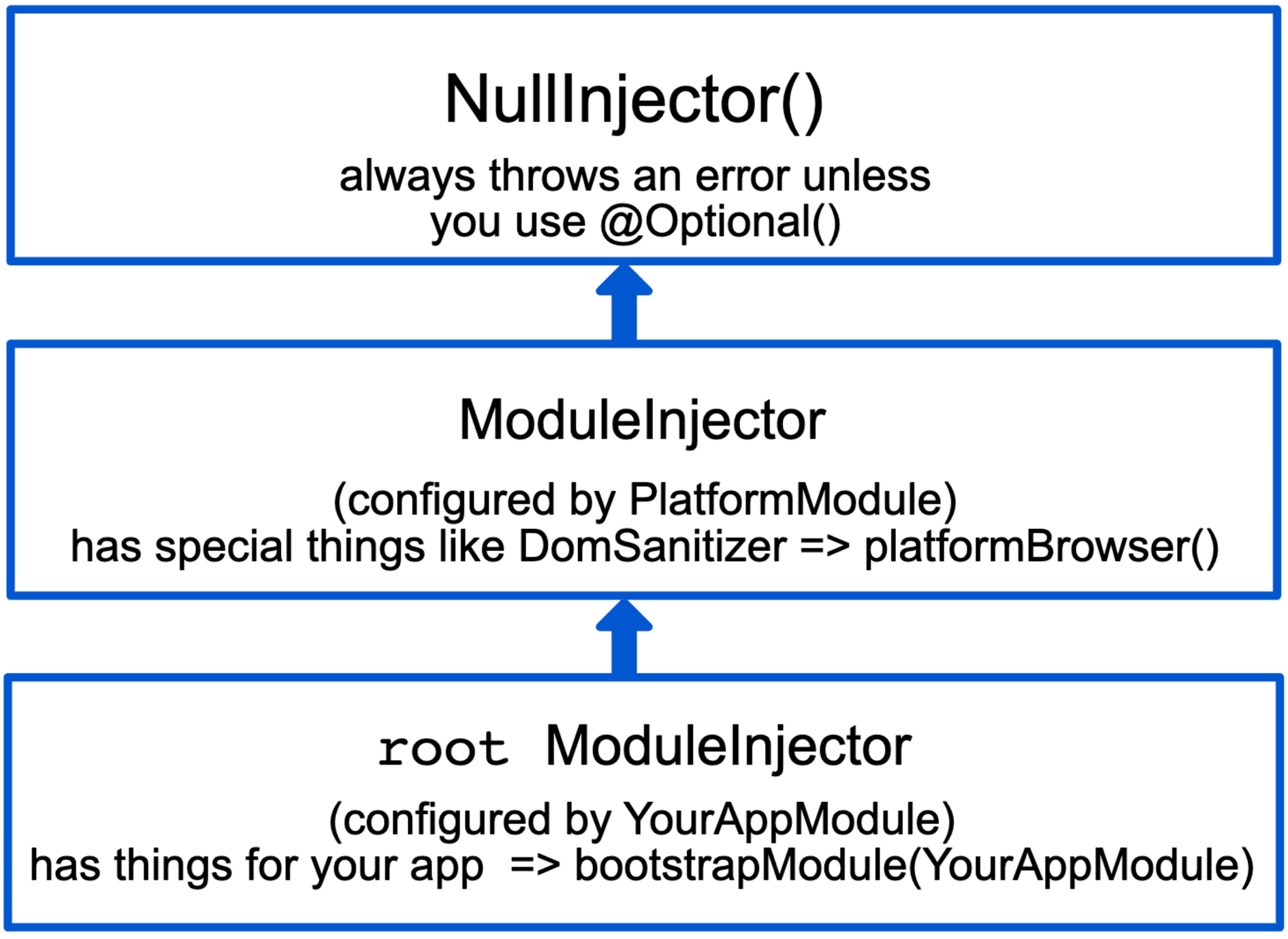
Option	Description
`providedIn?`	Determines which injectors will provide the injectable, by either associating it with an `@NgModule` or other `InjectorType`, or by specifying that this injectable should be provided in one of the following injectors: • 'root' : The application-level injector in most apps. • 'platform' : A special singleton platform injector shared by all applications on the page. • 'any' : Provides a unique instance in each lazy loaded module while all eagerly loaded modules share one instance.

Osservazioni

- Utilizzare la property `providedIn` di `@Injectable()` è preferibile a providers array di `@NgModule()`, perché in questo caso il tool di ottimizzazione può eseguire il *tree-shaking* che rimuove servizi che l'app non sta usando, con il vantaggio finale di ottenere dei bundle di dimensioni ridotte

- E' molto importante notare che, se un elemento viene configurato tramite `@Injectable(providedIn: 'any')`, l'inizializzazione sarà singleton per i moduli *eager loaded*, mentre nel caso di moduli *lazy initialized* questo comporterà una nuova reinizializzazione, ovvero, verrà generato un nuovo oggetto

Injectors hierarchy



Task Asincroni

- **Promise**
  - non sono lazy
  - pipeline unica
  - difficilmente annullabili
- **Observable**
  - lazy (necessitano del *subscribe*)
  - pipeline multipla
  - annullabili
  - offre operatori array-like (via RxJS)(`map`,`filter`,`forEach`,..)
  - possono essere creati da più sorgenti (es. eventi)

Observable vs Promise

- Entrambi servono a gestire task asincroni, ma:
  - un Promise
    - rappresenta un unico evento
    - il suo oggetto di ritorno è a sua volta un Promise
    - non è cancellabile
    - può comportare la terminazione di un'operazione con *success* o con una *failure*

Observable vs Promise

- un Observable
  - è uno stream, ovvero una concatenazione di eventi che permette di associare a ciascuno di questi una callback
  - il suo valore di ritorno non è un Observable
  - è cancellabile
  - offre più funzionalità rispetto al Promise per mezzo di operatori dedicati (map(),)
  - necessita di essere eseguito da un client per mezzo di una subscribe
  - ogni Observable è riconducibile ad una Promise (e **non viceversa**)

HttpClient

- È un servizio built-in di Angular, introdotto a partire da Angular 4.3
- Serve ad effettuare chiamate verso endpoint RESTFul tramite HttpClient, attraverso i metodi `get, post, put, delete, patch, head, jsonp`
- Va usato nei Service e non nelle Componenti

HttpClient con Observable

- Component:

```
this.bookService.findAll().subscribe(data => {
        this.books = data;
      }, error => {
        console.log(`Component Error: ${JSON.stringify(error)}`);
      }
);
```

- Service:

```
findAll(): Observable<Book[]> {
      return this.httpClient
        .get<Book[]>(apiURL)
        .pipe(
          retry(3),
          catchError(this.handleError)
        );
    }
  }
```

HttpClient

```
private handleError(error: HttpErrorResponse) {
  if (error.error instanceof ErrorEvent) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error.message);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong,
    console.error(
      `Backend returned code ${error.status}, ` +
      `body was: ${error.error}`);
  }
  // return an observable with a user-facing error message
  return throwError(
    'Something bad happened; please try again later.');
};
```

HttpClient con Promise

- Component:

```
this.bookService.findAll().then(data => {
        this.books = data;
      }, error => {
        console.log(`Component Error: ${JSON.stringify(error)}`);
      }
);
```

- Service:

```
findAll(): Promise<Book[]> {
    let promise = new Promise<Book[]>((resolve, reject) => {
      this.httpClient
        .get(apiURL)
        .toPromise()
        .then(
          (res) => {
            // Success
            resolve(res as Book[]);
          },
          (msg) => {
            // Error
            reject(msg);
          }
        );
    });
    return promise;
  }
```

Http Interceptors

- Con *HttpClient* sono stati introdotti gli **HTTP Interceptor,** che consentono di intercettare le *response* e le *request* delle chiamate HTTP effettuate nelle *Single Page Application*

- Lo scopo è di modificare, quando necessario, gli header HTTP dinamicamente (es.: aggiungere il 'Bearer token')

HttpInterceptor

```
import {
    HttpEvent,
    HttpInterceptor,
    HttpHandler,
    HttpRequest
} from '@angular/common/http';
import { Observable } from 'rxjs';

export class MyInterceptor implements HttpInterceptor {
    intercept(req: HttpRequest<any>, next: HttpHandler)
                :Observable<HttpEvent<any>> {
                //...
                 return next.handle(req);
            }
}
```

Esempio: Token Interceptor

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { AuthService } from './auth/auth.service';
import { Observable } from 'rxjs/Observable';

  @Injectable()
export class HttpInterceptorService implements HttpInterceptor {
  constructor(public auth: AuthService) {}
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    request = request.clone({
      setHeaders: {
        Authorization: `Bearer ${this.auth.getToken()}`
      }
    });
    return next.handle(request);
  }
}
```

HttpInterceptor

```
@NgModule({
  declarations: [
  ..
   ],
   imports: [
  ..
   ],
   exports: [
  ..
   ],
  providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: HttpInterceptorService,
    multi: true
  }],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Routing

- L'Angular Router abilita la navigazione dell'utente da una view all'altra

- È basato sul modello di navigazione dei browser (link, avanti, indietro,..)

- Il modulo da importare nell'applicazione è `RouterModule`

- Il RouterModule va poi configurato con le rotte di navigazione custom (path-componente)

- Ogni applicazione ha un solo router

Routing

```
import { RouterModule, Routes } from '@angular/router';

const appRoutes:Routes = [{path:'', component:AppComponent}];

@NgModule({
  declarations: [
    AppComponent,
    PersonDetailComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

Route Informations

- E' molto spesso utile passare informazioni da una componente all'altra, durante la navigazione (es. propagare l'`id` di un elemento)

- L'oggetto da utilizzare in questi casi è l'`ActivateRoute`, che va usato, all'interno di un componente attraverso DI del

- Esistono più modi di trasmettere parametri attraverso questo servizio

Route Informations: QueryParams

- Attraverso i `QueryParams` si possono specificare parametri senza modificare il mapping degli URI

Component.html

```
<a href="#" routerLink="/page" [queryParams]="{'name': 'Pippo'}"> Page </a>
```

Component.ts

```
import { Router, ActivatedRoute, ParamMap } from '@angular/router';

constructor(
  private route: ActivatedRoute
) {}

ngOnInit() {
  this.route.queryParams.subscribe(params => {
    this.name = params['name'];
  });
}
```

Route Informations: snapshot

- Attraverso `snapshot` si possono specificare parametri attraverso gli URI

app.routing.module.ts

```
const routes: Routes = [
..
  { path: 'page/:name', component: PageComponent }
];


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

Component.html

```
<a href="#" routerLink="/page/Pippo" > Page </a>
```

Component.ts

```
import { Router, ActivatedRoute, ParamMap } from '@angular/router';

constructor(
  private route: ActivatedRoute
) {}

ngOnInit() {
  this.name = this.route.snapshot.params.name;
}
```

Route Informations: paramMap

- Attraverso `paramMap` si possono specificare parametri attraverso gli URI e utilizzando gli `Observables` (approccio Reactive)

app.routing.module.ts

```
const routes: Routes = [
..
  { path: 'page/:name', component: PageComponent }
];


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

Component.html

```
<a href="#" routerLink="/page/Pippo" > Page </a>
```

Component.ts

```
import { Router, ActivatedRoute, ParamMap } from '@angular/router';

constructor(
  private route: ActivatedRoute
) {}

ngOnInit() {
this.route.paramMap.subscribe(params => {
      this.name = params.get("name");
 });
}
```

Routing & AuthGuard

```
import { AuthGuard } from '../auth/auth.guard';

const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard]
  }
];

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

Routing & AuthGuard

```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable(
  providedIn: 'root'
)
export class AuthGuard implements CanActivate {
  constructor(public auth: AuthService, public router: Router) {}
  canActivate(): boolean {
    if (!this.auth.isAuthenticated()) {
      this.router.navigate(['login']);
      return false;
    }
    return true;
  }
}
```

JwtModule

- `JwtModule` è un modulo riutilizzabile contenente un `HttpInterceptor`, in grado di filtrare automaticamente le request client, apponendo il token `JWT` come campo `Authorization`

- Home-Page:

`https://github.com/auth0/angular2-jwt`

- Installazione dipendenza:

```
npm install --save @auth0/angular-jwt
```

- Configurazione del modulo `JWT` nel modulo principale dell'applicazione

- Protezione delle rotte interne di navigazione con meccanismo di `AuthGuard`

- Definizione pagina di Login e corrispondente rotta di navigazione

Query DOM

- Angular offre una serie di decoratori utili per individuare uno o più elementi discendenti di un componente

- Le annotazioni in questione, sono:

  - `@ViewChild`

  - `@ViewChildren`

  - `@ContentChild`

  - `@ContentChildren`

- Mentre i primi due permettono di cercare nello *Shadow DOM* (default, Angular Components), gli altri permettono di cercare nel *Light DOM* (Angular Directives, dichiarate nella componente)

- L'individuazione degli elementi viene fatta specificando un `selector`

@ViewChild

- Serve a configurare una View Query per ottenere **un riferimento all'oggetto** del DOM, corrispondente al selector specificato

- L'oggetto ottenuto sarà di tipo `ElementRef`

- Tramite l'oggetto in questione è possibile accedere all'oggetto nativo, conservato all'interno come `nativeElement`

- Se la vista del DOM cambia e il nuovo elemento child fa match con il selector, la proprietà verrà aggiornata

- Le View Queries vengono impostate prima che la callback `ngAfterViewInit` venga invocata

- Dal momento che lo scopo di questo decoratore è quello di effettuare l'injection di un elemento del DOM in un oggetto JavaScript, il suo utilizzo è esposto a rischi a livello di Security (`XSS` attacks)

- Per approfondimenti sui rischi di `XSS`, consultare:

`https://angular.io/guide/security`

@ViewChild

- Esempio tipico:

Component.html

```
<button #closeButton>Close</h2>
```

(template reference)

Component.ts

```
@ViewChild('closeButton')
 btn: ElementRef;

 ngAfterViewInit() { // RACCOMANDATO
   this.btn?.nativeElement.click();
 }
```

@ViewChild

- E' possibile usare `ngOnInit()` al posto di `ngAfterViewInit()`?
  - Se vogliamo essere certi che i riferimenti iniettati da `@ViewChild` siano presenti, dovremmo scrivere sempre il nostro codice di inizializzazione all'interno di `ngAfterViewInit()`
  - In base al caso specifico, i template reference potrebbero essere presenti in `ngOnInit()`, ma non è sempre così

Selectors supportati

- Qualsiasi classe con decoratore `@Component` o `@Directive`
- Una variabile template reference come stringa (es. query: `<my-component #cmp></my-component>` with `@ViewChild('cmp'))`
- Qualsiasi provider definito nell'alberatura delle componenti figlio, partendo da un nodo parent (es: `@ViewChild(SomeService) someService!: SomeService`)
- Qualsiasi provider definito attraverso token string (es: `@ViewChild('someToken') someTokenVal!: any`)
- Un `TemplateRef` (es. query: `<ng-template></ng-template>` with `@ViewChild(TemplateRef) template;)`

@ViewChildren

- Serve a configurare una View Query, per ottenere un insieme di elementi o direttive, a partire dal DOM HTML

- L'oggetto restituito è di tipo `QueryList<T>`

Selectors supportati

- Qualsiasi classe con decoratore `@Component` o `@Directive`
- Una variabile template reference come stringa (es. query: `<my-component #cmp></my-component>` with `@ViewChildren('cmp')`)
- Qualsiasi provider definito nell'alberatura delle componenti figlio, partendo da un nodo parent (es: `@ViewChildren(SomeService) someService!: SomeService`)
- Qualsiasi provider definito attraverso token string (es: `@ViewChildren('someToken') someTokenVal!: any`)
- Un `TemplateRef` (es. query: `<ng-template></ng-template>` with `@ViewChildren(TemplateRef) template;`)
- In aggiunta rispetto a `@ViewChild`, si possono specificare selettori multipli, separandoli con la virgola (es. `@ViewChildren('cmp1,cmp2')`)

@ViewChildren

- Esempio completo:

Component.html

```
<div #div>Div 1 : #div is a template variable</div>
<div #div>Div 2 : #div is a template variable</div>
```

(template reference)

Component.ts

```
 @Component({
  selector: 'my-app',
  ..
})
export class App {
  @ViewChildren("div") divs: QueryList<ElementRef>

  ngAfterViewInit() {
    this.divs?.forEach(div => console.log(div));
  }
}
```

@ContentChild

- Serve a configurare una Content Query
- Utilizzato per ottenere il primo elemento o la direttiva che fa match con il selector del content DOM (`<ng-content>..</ng-content>`)
- Supporta gli stessi selettori di `@ViewChild`

@ContentChildren

- Serve a configurare una Content Query
- Usato per ottenere una QueryList di elementi o di direttive del content DOM (`<ng-content>..</ng-content>`)
- Supporta gli stessi selettori di `@ViewChildren`

Pipes

- Rappresenta l'equivalente dei Filtri in Angular JS

- Un Pipe è un processatore di dati in input

- Esistono Pipe built-in nella piattaforma Angular

- È sempre possibile creare dei Custom Pipe

Pipes built-in

- Pipe built-in:
  - DatePipe
  - UpperCasePipe
  - LowerCasePipe
  - CurrencyPipe
  - JsonPipe
  - ..
- Opzionalmente, si possono anche passare parametri alle singole pipe
- È anche possibile collegare più pipe in sequenza tra loro

Custom Pipe

- Una pipe è una classe decorata con `@Pipe` decorator (da Angular core library) e metadata

- È possibile ricorrere ad Angular CLI per la creazione di custom pipes:

```
ng generate pipe <custom_pipe_name>
```

- Tramite questo decorator è possibile specificare il nome della pipe da richiamare con la *template syntax*

- Una classe Pipe custom implementa il metodo `transform(..,..)` dell'interfaccia `PipeTransform,` che accetta un valore in input seguito da parametri opzionali e restituisce il valore trasformato

- Ci saranno parametri aggiuntivi, tanti quanti gli argomenti previsti dalla custom pipe

Credits

Dott. Ing. Luigi Brandolini

Software Engineer, Web-Enterprise Applications Specialist

Contacts:

E-Mail: luigi.brandolini@gmail.com

Skype: luigi.brandolini