Introduzione ad Angular

Un'applicazione Angular2 è composta da template HTML con markup Angularized, classi di componenti per gestire tali modelli, servizi per la logica dell'applicazione. Componenti e servizi vengono incapsulati in moduli.

Ambiente di sviluppo

Per semplificare l'attività di setup dell'ambiente di sviluppo, il team di Angular ha sviluppato Angular CLI, un ambiente a riga di comando per creare la struttura di un'applicazione Angular 2 già configurata secondo le linee guida ufficiali. Angular CLI è basato su **Node.js**, per installarlo:

```
npm install -g angular-cli
```

Una volta installato l'ambiente, per creare una nuova applicazione Angular 2 è sufficiente digitare il seguente comando:

```
ng new miaApp
```

Il comando crea la cartella in cui è impostata la struttura dell'applicazione. Qui si trovano, fra gli altri, i file di configurazione fondamentali:

- **package.json:** riferimenti alle dipendenze del progetto, incluso quello alle librerie di Angular 2
- **tslint.json:** configurazione di TSLint, un code checker per TypeScript
- **angular- cli.json:** configurazione del progetto dal punto di vista di Angular CLI
- **e2e:** contiene i test end to end di Protractor
- **node_modules:** contiene i moduli nodejs necessari per il setup e l'esecuzione dell'applicazione, inclusi i pacchetti di Angular
- **src:** qui sta il codice sorgente dell'applicazione.

Con il seguente comando lanciato all'interno della cartella del progetto:

```
ng serve
```

è avviato un web server che consente di accedere all'applicazione all'indirizzo **http://localhost:4200**. Inoltre è già disponibile il cosiddetto live reloading, la compilazione e il caricamento automatico dell'applicazione in seguito a modifiche ai file sorgenti. Inoltre, Angular CLI, è d'aiuto anche nella generazione di porzioni di codice.

Struttura di un'applicazione

La cartella **src** contiene la pagina **index.html**, l'elemento iniziale dell'applicazione.
Qui vi è un'unica direttiva Angular, rappresentanta dal tag **<app-root>**
Per il resto non c'è alcun riferimento a script da caricare poiché il markup per il caricamento degli script e dei fogli di stile è iniettato a runtime da Angular-CLI al momento della compilazione dell'applicazione.
Il punto d'ingresso dell'applicazione è il modulo **main** *(main.ts)* in cui sono definite le dipendenze principali ed è avviata l'applicazione:

```
// Polyfill: codice che fornisce servizi che non fanno parte di un browser web
import './polyfills.ts';

// Le funzionalità di Angular vengono importate dalla cartella di sistema @angular
```

```
// platform-browser-dynamic contiene funzionalità specifiche per il bootstrap
dell'applicazione all'interno di un browser
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
// core contiene le funzionalità di base di Angular
import { enableProdMode } from '@angular/core';
// codice di supporto alla gestione del progetto
import { environment } from './environments/environment';
// AppModule rappresenta il root module: modulo di partenza dell'applicazione
import { AppModule } from './app/';

if (environment.production) {
  // Abilita la modalità di produzione (in alternativa a quella di debug)
  enableProdMode();
}

// Il root module viene caricato ed avviato invocando il metodo bootstrapModule()
platformBrowserDynamic().bootstrapModule(AppModule);
```

Il root module

Un modulo è un contenitore di funzionalità che consente di organizzare il codice di un'applicazione. Un modulo TypeScript / ES6 è un modo per incapsulare, raggruppare, riutilizzare, distribuire e caricare codice JavaScript. In pratica si descrivono in un unico luogo tutte le dipendenze e gli elementi dell'applicazione.
Generalmente, un modulo contiene codice che incapsula una specifica funzionalità. Il modulo espone questa funzionalità al resto dell'applicazione definendo una serie di esportazioni che altre parti del sistema possono poi importare. Il modulo di base Angular 2, ad esempio, si chiama 'angular2/core' e dà accesso alle primitive Angular 2 di base come i componenti.
Sintatticamente, un modulo Angular 2 non è altro che una classe cui è stato applicato il decoratore **@NgModule**. Un decoratore è una funzione che arricchisce la classe di specifiche funzionalità, secondo lo standard introdotto dalle specifiche ECMAScript 2016 (ES7) e recepite da TypeScript, e fornisce metadati che indicano come compilare ed eseguire il codice del modulo.
Un'applicazione può essere composta di più moduli ma dev'esserci **un solo modulo di partenza o root module**, per convenzione chiamato **AppModule**.

L'app.module.ts creato da Angular-Cli:

```
// dipendenze da altri moduli
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

// I componenti del modulo da importare
import { AppComponent } from './app.component';

@NgModule({
    // lista di componenti ed altre funzionalità che appartengono al modulo corrente
    declarations: [ AppComponent ],
    // lista di elementi importati da altri moduli
    imports: [ BrowserModule ],
    // lista di oggetti di cui ci si vuole servire
```

```
    providers: [],
    // componente da utilizzare come root component dell'applicazione
    bootstrap: [ AppComponent ]
})

export class AppModule { }
```

Il modulo principale è sufficiente in una semplice applicazione con pochi componenti. Per applicazioni complesse conviene organizzare le diverse funzionalità in moduli che rappresentano le collezioni di funzionalità correlate. Infine s'importano questi moduli nel modulo principale. Oltre alla funzione organizzativa del codice i moduli Angular hanno un ruolo anche per la gestione della dependency injection, della compilazione e del lazy loading.

I componenti

I componenti Web consentono agli sviluppatori di ottenere il controllo totale sulla loro pagina web, fornendo modelli altamente funzionali che vengono incapsulati all'interno di selettori HTML personalizzati.
Un componente Angular controlla una parte del template chiamata View. Essa è una classe TypeScript cui vengono applicati uno o più decoratori. Il decoratore **@Component**, specifico del linguaggio TypeScript, permette di assegnare alla classe specifiche caratteristiche o metadati. Ad esempio, il componente ***AppComponent***, cui il modulo ***AppModule*** fa riferimento come root component dell'applicazione, è definito nel file **app.component.ts**.

```
// Importazione della classe Component e relativo decoratore
import { Component } from '@angular/core';

// Il decoratore arricchisce la classe con le funzionalità richieste mediante una
serie di meta-data
@Component({
    // indica l'elemento del markup cui è agganciato il componente
    selector: 'app-root',
    // il file HTML che descrive il markup del componente
    templateUrl: './app.component.html',
    // lista dei file CSS da applicare al markup
    styleUrls: ['./app.component.css']
})

// La classe AppComponent (La clausola export rende il componente accessibile da
altri componenti mediante la corrispettiva clausola import)
export class AppComponent {
    // proprietà della classe. In questo caso, title che descrive il testo da
visualizzare sullo schermo
    title = 'app works!';
}
```

Creare componenti con Angular-CLI

Sebbene la definizione di un componente sia relativamente semplice, i decoratori possono far riferimento a file esterni come template, i fogli di stile ed eventuali altre

risorse. Angular-CLI fornisce un comando specifico per la creazione della struttura di un componente. Il comando:

```
ng generate component miocomponente
```

crea una cartella con quanto occorre per la definizione del componente:

- **miocomponente.component.ts:** contiene la classe TypeScript del componente
- **miocomponente.component.html:** contiene il template del componente
- **miocomponente.component.css:** il codice CSS da applicare al componente
- **miocomponente.component.spec.ts:** codice di base per la definizione degli unit test.

Poiché un'applicazione Angular 2 è una composizione gerarchica di componenti, a questa struttura di base è possibile associare altri componenti.

Passaggio di dati tra componenti

Se un componente nidificato vuole ricevere input dal suo contenitore, deve esporre una proprietà a quel contenitore utilizzando il decoratore **@Input** importato dall'apposito modulo:

```
@Input() myProperty:string;
```

Nel componente contenitore, occorre definire la proprietà che si vuole passare al componente nidificato:

```
childProperty:string = 'Valore da passare al child';
```

Se il componente nidificato vuole inviare informazioni al suo contenitore, l'unico modo consiste nel generare un evento e utilizzarlo per passare i dati grazie al decoratore **@Output**. Poiché in Angular 2, un evento è definito con un oggetto **EventEmitter**, l'output prevede la creazione di un'istanza di EventEmitter:

```
@Output() myEvent: EventEmitter<string> = new EventEmitter<string>();
```

Il componente parent riceve evento e dati mediante Event Binding:

<child-selector (myEvent)='onMyEvent()'></child-selector>

Data Binding

Il processo di coordinamento dei valori dei dati avviene dichiarando associazioni tra fonti ed elementi di destinazione HTML. Ci sono quattro tipi di associazione dati:

- **Interpolazione:** mostra il valore del componente all'interno dei tag
- **Property Binding:** passa la proprietà dall'elemento padre al figlio
- **Event Binding:** generato quando si richiama il metodo del componente

- **Two-way Binding:** lega proprietà ed eventi in un'unica notazione utilizzando la direttiva ngModel.

Gestione eventi

L'unico modo per conoscere che tipo di azione ha eseguito l'utente, è quello di associare un gestore di evento agli elementi interessati, come accade in JavaScript o in jQuery. È sufficiente indicare il nome dell'evento da monitorare e il nome del metodo della classe che gestisce l'evento. Ad esempio, per intercettare la pressione su un link:

```
<a (click)="apriURL()">Clicca qui</a>
```

I più comuni gestori di evento sono:

- (click) - click su di un elemento
- (mouseover) - mouse sopra all'elemento
- (keyup) - pressione e rilascio di un tasto della tastiera
- (keyenter) - pressione del tasto enter della tastiera
- (change) – cambio su un elemento

Direttive

Ci sono tre tipi di direttive:

- **Componenti**
- **Direttive di attributo**
- **Direttive strutturali**

Il **componente** è in pratica una direttiva con un template associato.
La **direttiva di attributo** cambia l'aspetto o il comportamento di un elemento: **NgStyle**, ad esempio, può cambiare diversi stili contemporaneamente.
Una **direttiva strutturale** cambia il layout del DOM aggiungendo e rimuovendo elementi: **ngIf**, **ngSwitch** e **ngFor**.
Le direttive strutturali utilizzano il carattere asterisco **(*)** come prefisso nel loro nome. Questo carattere rappresenta un'abbreviazione sintattica per indicare che il markup sotto il loro controllo è un <template> da utilizzare per la generazione degli elementi del DOM. Si può fare a meno del carattere * specificando esplicitamente il tag <template>:

```
<template [ngIf]="!solaLettura">
  <div >
<button>Salva</button>
  </div>
</template>
```

Equivale a:

```
<div *ngIf="!solaLettura">
  <button>Salva</button>
</div>
```

Gestione dei Form

Angular 2 propone due approcci alla costruzione e gestione di form:

- **Template Driven Form:** si basa prevalentemente sulla definizione di una form tramite markup, inclusi i criteri di validazione,
- **Reactive Form:** detto anche Model Driven Form, questo approccio prevede una definizione minimale del template tramite markup e sposta la logica di validazione all'interno della definizione del componente. Inoltre prevede una modalità alternativa di costruzione della form stessa.

Template Driven Form

Ipotizzando di avere un campo d'input e relativo button, si può scrivere:

```
<input #email placeholder="Inserisci la tua email">
```

La notazione **#email** crea una "variabile locale del template", che rappresenta un riferimento all'elemento. Grazie ad essa è possibile accedere, in tempo reale, al valore contenuto tramite notazione **nome.value** (nel caso di nomi composti si usa la notazione "snake_case"):

```
<button (click)="inviaDati(email.value)">Invia</button>
```

Volendo visualizzare immediatamente il valore inviato:

```
<input #email placeholder="Inserisci la tua email" (keyup)="0">
<button (click)="invioDati(email.value)">Invia</button>
<p>Valore inserito: {{email.value}}</p>
```

Il recupero del dato inviato, è gestito dal metodo della classe dell'ipotetico componente:

```
inviaDati(dato_Form) {
 alert('Dato recuperato ' + dato_Form);
}
```

Service

Un servizio è il meccanismo utilizzato per condividere funzionalità tra i Componenti. Se una funzionalità è utilizzabile da molti componenti dell'applicazione, è buona pratica creare un unico servizio riutilizzabile. Quando un componente ha bisogno di questo servizio è sufficiente iniettarvi il servizio stesso. I Service sono quindi un modo efficace di trattare con le chiamate di dati asincroni, per la condivisione di dati tra componenti. Anche per creare un Service, come per componenti e moduli, è necessario importare la classe opportuna (Injectable) e usare il decoratore omonimo per definire la propria classe:

```
import {Injectable} from 'angular/core';

@Injectable()
export class MyService {
   …
```

```
}
```

Per utilizzare il Service è necessario fare due cose:

1. Fornire il servizio: il servizio viene cioè istanziato e messo a disposizione del sistema di dependency injection di Angular 2.
2. Iniettarlo dove occorre.

Routing

Angular 2 consente di dividere l'applicazione in diverse View attraverso cui navigare mediante il concetto di routing. Routing consente di indirizzare l'utente verso diversi componenti in base alla URL digitata nel browser o gli appositi link cliccati.
I componenti e le classi principali di routing sono:

routerLink: La direttiva router-link è utilizzata per dichiarare i collegamenti. Può contenere anche parametri opzionali:

```
<a routerLink="/alias_componente"> Click </a>
// L'alias del componente può essere definito nel file module.ts
```

router-outlet: funge da segnaposto per le view che verranno renderizzate:

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

RouterModule: Definito nel file module.ts , mappa gli URL ai componenti:

```
RouterModule.forRoot([
      { path: 'uno', component: UnoComponent },
      { path: 'due', component: DueComponent },
      { path: 'tre', component: TreComponent }
    ])
```

Dependency Injection

Probabilmente uno dei più grandi e "moderni" problemi delle programmazione OOP è rappresentato dalle dipendenze. Se si desidera scrivere codice di buona qualità si dovrebbe limitare il più possibile gli effetti delle dipendenze tra le classi. Il pattern di progettazione **Dependency Injection** consente di iniettare gli oggetti in una classe, invece di crearli al suo interno.

```
class MyClass {
    private dipendenza;
    public function MyClass(arg1, arg2)   {
        this.dipendenza = new MyDipendenza(arg1, arg2);
    }
}

myIstanza= new MyClass('arg1', 'arg2');
```

In questo esempio, la classe MyClass è fortemente accoppiata alla sua dipendenza, la classe MyDipendenza. Una modifica al costruttore di MyDipendenza (l'aggiunta di un parametro per esempio), si ripercuoterà a cascata su qualsiasi punto del programma in cui si sono create istanze della classe MyClass.
Grazie alla DI, invece di creare/ottenere la dipendenza nel costruttore di MyClass gliela si inietta a runtime:

```
class MyClass {
    private dipendenza;
    public function MyClass(dipendenza)   {
        this.dipendenza = dipendenza;
    }
}

myIstanzaDipendenza = new MyDependency('arg1', 'arg2');
myIstanzaClasse = new MyClass(myIstanzaDependency);
```

Ora, la configurazione dell'oggetto MyDipendenza è semplice, ed anche sostituirlo con un'altra classe è molto facile, tutto è modificabile senza "toccare" l'istanza di MyClass.

Angular dispone di un proprio framework per l'iniezione di dipendenza.
Quando Angular crea un componente, chiede un injector per le prestazioni richieste dal componente. Un Iniettore mantiene un contenitore di istanze di service che ha creato in precedenza. Se un'istanza del servizio richiesto non è nel contenitore, l'iniettore ne crea uno e lo aggiunge al contenitore prima di fornire il servizio ad Angular. Quando tutti i servizi richiesti sono stati risolti e forniti, Angular può chiamare il costruttore del componente con quei servizi come argomenti. Questa è la Dependency Injection.
Quindi è necessario aver già registrato un provider del Service nell'iniettore. Un provider può creare o restituire un servizio, in genere la classe del servizio stesso. È possibile registrare i provider nei moduli o nei componenti. In genere, si aggiunge il provider nel modulo principale in modo che la stessa istanza di servizio sia disponibile ovunque. In alternativa, è possibile registrare il service a livello del componente nella proprietà providers dei metadati del @Component. Naturalmente, la registrazione a livello di componente significa che si ottiene una nuova istanza del servizio per ogni nuova istanza di tale componente.

Ciclo di vita dei componenti

Ogni componente ha un ciclo di vita gestito da Angular.
Angular crea, renderizza il componente e i suoi figli, controlla quando le sue proprietà con associazione a dati cambiano, e lo distrugge prima di rimuoverlo dal DOM.
Angular offre lifecycle hooks che forniscono visibilità in questi momenti chiave e la capacità di agire quando si verificano. Una direttiva ha lo stesso insieme di lifecycle hooks, tranne quelli sono specifici per il contenuto dei componenti e delle viste.
Gli sviluppatori possono accedere ai momenti chiave del ciclo di vita, mediante l'implementazione di una o più delle interfacce Lifecycle fornite da Angular. Ogni interfaccia ha un unico metodo il cui nome è il nome dell'interfaccia con prefisso ng.
Dopo la creazione di un componente/direttiva tramite costruttore, Angular chiama i metodi hooks del ciclo di vita nella seguente sequenza in momenti specifici:

ngOnChanges
Risponde quando angolare reimposta le proprietà di ingresso associate ai dati. Il metodo riceve un oggetto SimpleChanges dei valori delle proprietà attuali e precedenti.
Chiamato prima di ngOnInit e ogni volta che una o più proprietà di input associati ai dati cambia.

ngOnInit
Inizializza la direttiva/componente dopo che Angular espone le proprietà associate a dati e imposta le proprietà di input della direttiva/componente.
Chiamato una volta, dopo il primo ngOnChanges.

ngDoCheck
Rileva e agisce sulle modifiche che Angular non può o non vuole rilevare da solo.
Chiamato ad ogni rilevamento delle modifiche, subito dopo ngOnChanges e ngOnInit.

ngAfterContentInit **
Risponde dopo che Angular proietta contenuti esterni nella vista del componente.
Chiamato una volta dopo la prima NgDoCheck.

ngAfterContentChecked **
Risponde dopo che Angular controlla il contenuto proiettato nel componente.
Chiamato dopo ngAfterContentInit e ad ogni successivo NgDoCheck.

ngAfterViewInit **
Risponde dopo che Angular inizializza la vista del componente e dei figli.
Chiamato una volta dopo la prima ngAfterContentChecked.

ngAfterViewChecked **
Risponde dopo che Angular controlla le viste del componente e dei figli.
Chiamato dopo ngAfterViewInit e ogni successivo ngAfterContentChecked.

ngOnDestroy
Pulisce poco prima che Angular distrugga la direttiva / componente. Disattiva Observable e gestori di eventi al fine di evitare perdite di memoria.
Chiamato poco prima che Angular distrugga la direttiva / componente.

*** Hook attivo solo per i componenti.*