Teoria Frontend

Contrader

18 settembre 2023

Indice

Parte I

Front-End

Capitolo 1

TypeScript

In questo capitolo introduciamo Javascript, affrontandone la trattazione assumendo una buona conoscenza di base di Java. Ci concentreremo quindi sulle differenze tra i due linguaggi e sui punti più cruciali, lasciando che siano gli snippet di codice inseriti a far prendere familiarità con le basi e la sintassi.
Vedremo in seguito il typescript, un'estensione (superset) di javascript che introduce la tipizzazione. Anticipiamo che il comportamento a runtime di javascript e typescript è identico, di conseguenza nella seconda parte utilizzeremo ciò che abbiamo appreso su Javascript e vederemo come si traduce in typescript.

1.1 Javascript

Una delle cose più sorprendenti di JavaScript (che d'ora in poi indicheremo solamente con JS per brevità) è che nonostante sia un linguaggio orientato a oggetti, in esso non è presente il concetto di classe (almeno fino a ECMAScript 6). Consideriamo ad esempio il seguente programma:

```
var eroe = {};

eroe.nome = "ContraderMan";
eroe.forza = 100;
```

Quello che abbiamo appena fatto è stato inizializzare un oggetto (eroe) e definirne due proprietà: nome e forza. In JS non abbiamo una dichiarazione tipizzata come in java (ovvero per dichiarare un oggetto non dobbiamo specificare di che tipo questo oggetto sia), per dichiararre una variabile utilizziamo solamente la keyword var. **NOTA BENE:** la tipizzazione è comunque presente ma è JS stesso a capire con che tipo di dati ha a che fare e a prendere gli accorgimenti necessari. JS è quindi un linguaggio weakly typed.
In un certo senso, è utile pensare a un oggetto di JS come a una mappa <String, Object> Java, ad esempio quanto appena visto si potrebbe pensare come:

```
Map<String, Object> eroe = new HashMap<>();

eroe.put("nome", "ContraderMan");
eroe.put("forza", 100);
```

1.1.1 JS Functions

Un concetto nuovo che JS introduce rispetto a Java è quello di Function. In JS una funzione è semplicemente un valore di tipo Function. Riprendiamo l'esempio precedente

e creiamo una funzione che assegnamo a una variabile myFunction. Settiamo poi una nuova proprietà dell'oggeto eroe utilizzando questa variabile.

```
var myFunction = function() {
     console.log("La mia funzione");
}

eroe.parla = myFunction;
```

Possiamo chiamare delle function come se fossero dei metodi Java, ad esempio per chiamare la funzione definita sopra scriveremo semplicemente eroe.parla().
La cosa che più si avvicina a questo approccio in Java è l'utilizzo di un'interfaccia funzionale. Ad esempio, una possibile interpretazione di quanto appena fatto in Java potrebbe essere:

```
 public interface Power {
     void use();
 }

 public class Eroe {

     private Power power;

     public void setPower(Power power) {
        this.power = power;
     }

     public parla() {
         this.power.use();
     }

}

 public static void main(String[] args) {

     Eroe eroe = new Eroe();

     eroe.setPower(
         () -> S.out.println("Nella mia funzione");
     )

     eroe.parla();
 }
```

Vediamo ora come cambia l'utilizzo della keyword this: saremo sorpresi di scoprire che JS ci permette di fare delle cose abbastanza carine rispetto a Java.
Iniziamo con un esempio:

```
var superMan = {

     heroName: "SuperMan",

     sayHello: function() {

         console.log("Ciao, mi chiamo " + this.heroName );

     }
```

```
};

superMan.sayHello();
```

Iniziamo con l'osservare come abbiamo dichiarato direttamente le proprietà dell'oggetto superMan: tramite coppie (nome,valore) separati da due punti. Questo tipo di notazione è detta JavaScript Object Notation, per gli amici JSON. In secondo luogo notiamo come abbiamo referenziato la variabile heroName all'interno dello stesso oggetto, in maniera analoga a quanto faremmo in Java. Ma che succede se tramandiamo la funzione a un oggetto che non ha nessuna variabile heroName al suo interno? Ad esempio:

```
var nuovaFunzione = superMan.sayHello;

nuovaFunzione();
```

In questo caso otterremmo in output "Ciao, mi chiamo undefined". Per ovviare a questa problematica, in JS possiamo passare il contesto a cui deve fare riferimento la keyword this tramite il metodo .call, ad esempio se avessimo scritto:

```
var spiderMan = {
    heroName: "Spiderman"
};

var nuovaFunzione = superMan.sayHello;

nuovaFunzione.call(spiderman);
```

avremmo ottenuto in putput "Ciao, mi chiamo Spiderman", in quanto la keyword this va a cercare all'inteno dell'oggetto spiderman la proprietà ad essa associata.

1.1.2 Ereditarietà in JS

In JS non abbiamo ereditarietà a livello di classi: gli oggetti possono estendere direttamente altri oggetti. Più nel dettaglio, ogni volta che definiamo un oggetto questo avrà una proprietà implicita che punta a un 'oggetto padre'. Tale proprietà si chiama __proto__ e l'oggetto a cui punta è detto il prototype. Per questo motivo, questo tipo di ereditarietà viene chiamata Prototypical Inheritance. Vediamo come funziona: quando chiamiamo una proprietà di un oggetto, JS la cercherà inizialmente nell'oggetto stesso e, se non la trova, proverà nel prototype, poi nel prototype del prototype e così via. Vediamo un esempio per capire meglio:

```
var supereroe = {

     editor: "EditorSerioSRL"

};

var spalMan = {};

spalMan.__proto__ = supereroe;

console.log(spalMan.editor);      //stampa EditorSerioSRL
```

Il concetto di ereditarieta prototipica permette di svolgere in JS qualsiasi task che si può svolgere in Java tramite l'ereditarietà classica. Ma che succede coi costruttori?

1.1.3 Costruttori e funzioni costruttore

In JS possiamo ritrovare un tentativo di rendere la creazione di un oggetto simile a JAVA, nella forma delle funzioni costruttore. Una funzione costruttore definisce nella sua stessa dichiarazione un oggetto, specificandone le proprietà e può essere chiamata tramite la keyword new alla stessa maniera di un costruttore in Java. Ad esempio

```
function SuperHero(nome, forza) {
     this.nome = nome;
     this.forza = forza;
}

var contraderman = new SuperHero("Alberto", 169);

console.log(contraderman.nome);     //Stampa "Alberto"
```

Questa sintassi è tuttavia **sconsigliata**: supponiamo ad esempio di voler far si che tutti i supereroi abbiano un metodo sayHello() al loro interno. Ad esempio, lo potremmo inserire in un prototype object degli oggetti SuperHero tramite la sintassi FunzioneCostruttore.**prototype**.nomeMetodo, come di seguito

```
SuperHero.prototype.sayHello = function() {
     console.log("Ciao, il mio nome e' " + this.nome);
}
```

Tuttavia questa soluzione è artificiosa e di difficile lettura, inoltre non assomiglia per nulla a Java! Per questo motivo si preferisce accantonare del tutto l'utilizzo delle funzioni costruttore in favore di Object.create. Vediamo subito un esempio per capire di che si tratta:

```
var superHeroPrototype = {

     sayHello: function() {
         console.log("Ciao, il mio nome e' " + this.nome);
     }

};

var superman = Object.create(superHeroPrototype);
superman.nome = "Superman";
```

Per maggiori dettagli su come utilizzare a pieno questo meccanismo rimandiamo a questo articolo.

1.1.4 Closure e Lambda

In JS, una Closure non è molto diversa da una funzione lambda di Java. Consideriamo il seguente esempio (in Java):

```
public interface FlyCommand {
     public void fly();
}

public class EoreVolante {
```

```
        private String name;

        public EroeVolante(String name) {
            this.name = name;
        }

        public void fly(FlyCommand command) {
            command.fly();
        }
    }

    public static void main(String[] args) {
        String destinazione = "Marte";

        EroeVolante superMan = new EroeVolante("Superman");

        superMan.fly(
            () -> S.out.println("In volo verso " + destinazione)
        );
    }
```

In Java, ogniqualvolta referenziamo una variabile all'interno di una lambda, questa deve essere (effectively) final, ovvero non deve cambiare. Se ad esempio nel main avessimo scritto

```
    String destinazione = "Marte";

        EroeVolante superMan = new EroeVolante("Superman");

        superMan.fly(
            () -> S.out.println("In volo verso " + destinazione)
        );

        destinazione = "Terra";
```

Avremmo ottenuto il seguente errore:
java: local variables referenced from a lambda expression must be final or effectively final.
Ciò non accade con le closure di JS. Quella che a prima vista può sembrare una differenza irrilevante è in realtà una feature molto potente, in quanto ci consente di creare dei moduli incapsulati anche senza avere a disposizione i modificatori d'accesso che ci fornisce Java. Consideriamo ad esempio:

```
    function createHero(nome) {

        var heroName = nome;

        return {

            fly: function(destination) {
                console.log(heroName + "in volo per" + destination);
            }

        }

    }
```

In questo caso di fatto l'unico modo per accedere alla variabile heroName è attraverso la funione fly, la quale di fatto protegge l'accesso a heroName alla stessa maniera di un

getter.
Ad esempio

```
var superman = createHero("Superman");

superman.fly("Marte");
console.log("Il nome da eroe e' " + superman.heroName);
```

avremo in ordine ”Superman in volo verso Marte” e ”Il nome da eroe è undefined”. Una funzione come createHero che incapsula delle informazioni accessibili solo tramite delle interfacce (che sono restituite come Function nel return) è detta modulo.

1.1.5 Scope delle variabili

Un altra differenza da ricordare è l’assenza di uno scope per le variabili. Ciò si traduce, ad esempio, nel fatto che una variabile utilizzata in un loop sia visibile anche ad di fuori di esso. Ad esempio:

```
function counterLoop() {

    console.log('counter before declaration = ' + i);

    for (var i = 0; i < 3 ; i++) {
        console.log('counter = ' + i);
    }

    console.log('counter after loop = ' + i);
}

counterLoop();
```

darà come output:

```
counter before declaration = undefined
counter = 0
counter = 1
counter = 2
counter after loop = 3
```

È interessante anche notare come alla linea 3 non abbiamo un errore: questo in quanto l’interprete scansiona l’intera funzione e salva la lista delle variabili al suo interno e solo dopo inizia a interpretare la funzione riga per riga. Per via di questo comportamento, è best practice dichiarare tutte le variabili che si andranno ad utilizzare all’inizio del codice, in modo da rendere il codice leggibile e da evitare sorprese.
NOTA: a partire da ECMAScript 6 sono presenti anche le keyword **let** e **const**, entrambe le quali ci consentono di dichiarare delle variabili con scope delimitato, con la seconda che consente anche di rendere tali variabili immutabili. A differenza delle variabili dichiarate ocn var, quelle dichiarate con let e const non sono disponibili fino a che la loro dichiarazione non è raggiunta nel codice, per questo motivo dichiarare tutte le variabili all’inizio del codice è ancora più consigliato.

1.1.6 Classi in JS

Come anticipato all'inizio, a partire da ECMAScript6 è stato introdotto anche in JS il concetto di classe. Queste sono di fatto delle 'funzioni speciali' che presentano due componenti principali nella sintassi: le dichiarazioni e le espressioni.
Per definire una classe tramite una dichiarazione utilizziamo la keyword class come segue:

```
class Rectangle {
     constructor(height, width) {
         this.height = height;
         this.width = width;
     }
}
```

Per istanziare una classe si può utilizzare la keyword new, alla stessa maniera di Java.
Un altro modo per definire una classe è tramite un'espressione, ad esempio

```
let Rectangle = class {      //o anche class Nome per assegnare un nome
    diverso dalla variabile
     constructor(base, altezza) {
         this.base = base;
         this.altezza = altezza;
     }
};
console.log()
```

Il metodo constructor è un metodo speciale, il quale permette di creare e inizializzare oggetti di questa classe. Il funzionamento di una classe in JS è pressocchè identico alle classi Java, rimandiamo a questo articolo per una descrizione completa.

1.2 TypeScript

TypeScript (che per brevità abbrevieremo con TS) è un'estensione di JS che introduce tipi, classi, interfacce e muùoduli opzionali. È importante ricordare che typescript usa lo stesso runtime di JS.

1.2.1 Tipi in TS

In Java, siamo abituati al fatto che ogni valore/oggetto sia di uno e un solo tipo. In TS ciò non è più vero: i tipi sono un insieme di valori che condividono qualcosa tra loro. Pensare ai tipi come degli insiemi ci permette di semplificare alcune operazioni, come ad esempio passare come parametro di un metodo un valore che può essere o una stringa o un intero, in quanto non esiste un tipo che rappresenta un tale valore.
In TS ciò diventa naturale appena realizziamo che pensare ai tipi come adegli insiemi ci permette di fare tutte le operazioni insiemistiche con tali tipi. Nell esempio precedente potremo indicare che il tipo del parametro deve essere l'unione del tipo/insieme stringa e del tipo/insieme number, tramite l'operatore | :

```
myMethod(param: string | number) { ... }
```

Un'altra differenza con quanto siamo abituati a fare in Java è che gli oggetti in TS non sono di un tipo preciso. Ad esempio, se costruiamo un oggetto che soddisfa una data interfaccia, possiamo usare tale oggetto ogniqualvolta una tale interfaccia è richiesta, anche se non abbiamo mai dichiarato esplicitamente una relazione tra l'oggetto e l'interfaccia. Ad esempio:

```
interface Pointlike {
   x: number;
   y: number;
}
interface Named {
   name: string;
}

function logPoint(point: Pointlike) {
   console.log("x = " + point.x + ", y = " + point.y);
}

function logName(x: Named) {
   console.log("Hello, " + x.name);
}

const obj = {
   x: 0,
   y: 0,
   name: "Origin",
};

logPoint(obj);
logName(obj);
```

Poichè la const obj soddisfa sia l'interfaccia PointLike che quella Named, possiamo passare obj come parametro sia di logPoint che di logName senza alcun problema.
I tipi in TS sono organizzati in maniera **strutturale** non **nominale**, ovvero sono le proprietà al loro interno a sancire le relazioni tra tipi, non le loro dichiarazioni. **obj** implementa di fatto entrambe le interfacce di sopra, nonostante nella sua dichiarazione ciò non figuri mai. (Se questo tipo di libertà vi piace, clickate qui :))
Alcune conseguenze della tipizzazione strutturale includono gli **empty types** e gli **identical types**.
Vediamo di che si tratta esaminando due esempi:

```
class Empty {}

function fn(arg: Empty) {
   [ ... ]
}

fn({ k: 10 });
```

In questo caso il codice è perdettamente valido in quanto l'oggetto k:10 ha tutte le proprietà di Empty, in uqanto Empty non ha proprietà!

```
class Car {
    drive() { //fa qualcosa }
}
class Golfer {
    drive() { //fa qualcos altro }
}
//assegnamento valido
let w: Car = new Golfer();
```

Questo è un caso di identical types; anche in questo caso la dichiarazione è corretta in quanto sia la classe Car che la classe Golfer hanno la stessa struttura. Per evitare errori, è buona norma utilizzare identical types solo quando questi sono effettivamente correlati.

1.2.2 Tipi comuni

In questa sezione ci occupiamo di descrivere i tipi più comuni che si incontrano nella stesura di un codice TS. I tre tipi primitivi di TS sono **string**, **number** e **boolean**. A differenza di Java, in TS non abbiamo diversi valori specifici per diversi tipi di numero, come ad esempio float o int, ma fa tutto parte di number.
Per indicare gli array si procede come in Java, semplicemente facendo seguire una coppia di parentesi quadre al tipo. Ad esempio, myArr : string[] dichiara che la variabile myArr è un array di stringhe. Un altro modo per dichiarare un array è utilizzare Array<T>, dove T è il tipo degli elementi dell'array. (Sì, i generici esistono anche in TS e il loro funzionamento è pressocchè identico a Java, per una guida dettagliata guardate qui)
Un altro tipo importante è **any**, che può essere utilizzato quando vogliamo evitare che un particolare valore causi errori di type-check. Quando dichiariamo un valore di tipo any, tutte le sue proprietà saranno in automatico di tipo any. Inoltre esso potrà inoltre essere chiamato come una function e assegnarlo a un altro valore di qualsiasi tipo. In pratica l'utilizzo di any disattiva ogni meccanismo di controllo dei tipi, ad esempio le seguenti righe di codice sono tutte valide:

```
let obj: any = { x: 0 };
 obj.foo();
 obj();
 obj.bar = 100;
 obj = "hello";
 const n: number = obj;
```

Vediamo ora come si applica la tipizzazione alle funzioni. In TS possiamo specificare sia il tipo degli argomenti che il tipo dell'output di una funzione. La sintassi è la seguente:

```
myFunction(arg1 : Type1, ... argN : TypeN): returnType { ... }
```

Nell'esempio precedente la sintassi è la stessa se al posto di un tipo primitivo si passano degli argomenti che hanno di tipo-oggetto, ad esempio

```
function stampaCoordinate( pt : { x: number; y: number}) {
     console.log("L'ascissa ha valore " + pt.x);
     console.log("L'ordinata ha valore " + pt.y);
}

stampaCoordinate({x: 3 , y: 4});
```

Per indicare che una proprietà di un oggetto è opzionale si fa seguire un punto di domanda al nome della proprietà. Ad esempio, se vogliamo che la coordinata z sia opzionale nell'oggetto del metodo precedente, scriveremmo:

```
function stampaCoordinate( pt : { x: number; y: number; z?: number}) {
     console.log("L'ascissa ha valore " + pt.x);
     console.log("L'ordinata ha valore " + pt.y);
     if( z!== undefined) {
         console.log("La cordinata z ha valore " + pt.z);
     }
}

stampaCoordinate({x: 3 , y: 4});
stampaCoordinate({x: 1, y: 2, z: 3});
```

Ogniqualvolta si opera con una proprietà opzionale bisogna sempre controllare che essa non sia undefined, come abbiamo fatto sopra.

Approfondiamo un attimo sull'unione di tipi. OCme anticipato prima, in TS possiamo indicare che un valore possa essere di un itpo oppure un altro utilizzando l'operatore di unione | . Quando lavoriamo con tipi unione, dobbiamo stare attenti a utilizzare solamente le proprietà che sono comuni ai due metodi. Se per esempio proviamo a scrivere

```
function stampaID(id: number | string) {
     console.log( id.toUpperCase() );
}
```

Otterremmo il seguente errore:
Property 'toUpperCase' does not exist on type 'string | number'.
Property 'toUpperCase' does not exist on type 'number'.
Una soluzione potrebbe essere quella di controllare il tipo effettivo dell'argomento, ad esempio tramite l'operatore typeof, ma in generale va bene una qualsiasi soluzione che consenta di chiamare proprità specifiche solo dopo essersi assicurati che l'oggetto si di tipo giusto.

```
function stampaID(id: number | string) {
     if ( typeof id === "string" ) {
         console.log( id.toUpperCase() );
     } else { console.log(id); }
}
```

Un concetto a prima vista inutile ma che nasconde un gran potenziale è quello di tipo letterale. Un tipo letterale indica che una variabile può far riferimento solamente a un valore specifico, ad esempio

```
let x: "hello" = "hello";
 // OK
 x = "hello";
 // ...
 x = "Buonasera";
 // avremo l'errore: Type '"Buonasera"' is not assignable to type '"hello"'.
```

L'esempio di sopra utilizzava delle stringhe, tuttavia i tipi letterali si possono utilizzare anche per number e boolean. Il potenziale nascosto dei tipi letterali è che utilizzando unioni di questi si può garantire che un qualcosa accetti solamente determinati valori. Ad esempio, possiamo garantire che una funzione accetti solamente degli input predefiniti:

```
function myFunction( libera: string, nonLibera: "destra" | "centro" | "
    sinistra") { ... }
myFunction("ciao", "destra"); //ok
myFunction("Hola", "chica"); //errore: Argument of type '"chica"' is not
    assignable to parameter of type '"destra" | "centro" | "sinistra"'.
```

Vediamo infine due tipi primitivi che vengono utilizzati per indicare un valore assente o non inizializzato: **null** e **undefined**. Il comportamento di entrambi questi tipi dipende dall'opzione di compilazione **strictNullChecks**. Se tale opzione è disattivata, si può accedere a valoripotenzialmente null/undefined e questi possono essere assegnati normalmente. Se invece tale opzione è attiva, bisognerà testare un valore potenzialmente null/undefined prima di poterci operare, come avevamo visto in precedenza per le proprietà opzionali. Per il controllo su null possiamo testare direttamente per uguaglianza con il tipo **null**, ad esempio

```
function doSomething(x: string | null) {
    if (x === null) {
      // do nothing
    } else {
      console.log("Hello, " + x.toUpperCase());
    }
}
```

In alternativa, TS ci mette a disposizione l'operatore ! per indicare che un valore non deve essere null/undefined. Tramite questo operatore, possiamo ad esempio scrivere:

```
function doSomething(x?: number | null) {
    console.log(x!.toFixed());
}
```

Capitolo 2

HTML e CSS

In questo capitolo affronteremo le basi di html e css, due tool fondamentali per lo sviluppo di una qualsiasi applicazione web in quanto responsabili della parte grafica e strutturale di questa.

2.1 HTML

HTML sta per HyperText Markup Language ed è il linguaggio di markup standard per la creazione di pagine web. Esso si occupa did escrivere la struttura di una pagina tramite una serie di elementi, i quali dicono al broswer come mostrare a schermo il contenuto.

2.1.1 Struttura

Ogni documento html ha una struttura ben precisa:

```
<!DOCTYPE html>
<html>
    <head>
        metadati e altre informazioni
        <title>Page Title</title>
    </head>
    <body>

        Contenuto della pagina

    </body>
</html>
```

1. il tag !DOCTYPE html dichiara che un documento è da trattarsi come un documento HTML5.
2. l'elemento html contiene tutto il documento: tutti gli altri elementi devono andare al suo interno.
3. l'elemento head contiene meta-dati relativi alla pagina. Un esempio di elemento contenuto al suo interno è title, che specifica il titolo della pagina in questione ed è visualizzato nella del broswer.
4. l'elemento body delimita il corpo della pagina. Al suo interno andranno quindi tutti gli elementi che si vogliono visualizzare, come ad esempio intestazioni, paragrafi, tabelle, link, immagini, liste etc.

2.1.2 Elementi HTML

Un elemento html è definito da un tag di apertura ed un tag di chiusura: tutto ciò che è contenuto tra questi due tag è l'elemento vero e proprio.

```
<tag> contenuto </tag>
ad esempio

<h1> Ciao mondo, questo  un heading </h1>
<p> Questo  un paragrafo </p>
<br>
```

Alcuni elementi, come l'elemento
 di sopra (che serve per andare a capo), noon hanno nè contenuto nè un tag di chiusura. Questi elementi sono detti empty elements.
Vediamo ora brevemente alcuni degli elementi più importanti:

- **div**: è un generico elemento html che viene solitamente utilizzato come contenitore per dividere il contenuto del body in più zone. Il tag associato è .
- **Headings**: sono definiti dai tag <h1>, <h2>, ... , <h6>, in ordine di importanza.
- **Paragrafi**: sono definiti dal tag <p>.
- **Links**: sono definiti dal tag <a>. Il tag di apertura contiene al suo interno l'attributo href, nel quale si specifica la destinazione del link. Gli attributi vengono utilizzati in generale per fornire informazioni aggiuntive su un elemento HTML, li vedremo più nel dettaglio a breve.

```
        <a href="http://www.sitosicuro.xyz/"> Clicka qui trust bro </a>
```

- **Immagini** sono definite dal tag <img>. Questo è un **empty element**. All'interno del tag di apertura troviamo gli attributi src, alt, width e heigth, i quali indicano rispettivamente la fonte dell'immagine, un testo da visualizzare qualora l'immagine non fosse disponibile, la larghezza e l'altezza dell'immagine. Questi attributi non devono per forza andare nell'ordine detto qui sopra.

```
        <img src="gattino.jpg" alt="Foto di un gattino" width="104" height
            ="142">
```

  L'attributo source può far riferimento sia a un'immagine hostata all'interno del sito sia a un URL esterno.

2.1.3 Attributi

Vediamo più nel dettaglio alcuni attributi e le best practices legate al loro utilizzo.

Style

L'attributo style è utilizzato per specificare lo stile grafico di un dato elemento. È buona norma utilizzarlo solamente in fase di test e specificare lo stile finale di una pagina all'interno di un foglio di stile apposito, come vedremo nella seconda parte di questo capitolo. Ad esempio, per stilizzare un paragrafo rendendo il testo rosso faremmo:

```
<p style="color:red;"> Il mio paragrafo rosso</p>
```

Lang

L'attributo lang viene utilizzato all'interno del tag <hmtl> per specificare la lingua della pagina. Questo non influisce su come la pagina viene visualizzata ma aiuta i motori di ricerca a visualizzare le pagine più rilevanti. Il linguaggio viene specificato attraverso delle abbreviazioni, come "en" per l'inglese o "it" per l'italiano.

```
<html lang="it">
 ...
</html>
```

L'attributo title

Viene utilizzato per definire informazioni aggiuntive su un elemento, che verranno visualizzate quando si passa sopra all'elemento col mouse:

```
<p title="Info aggiuntive"> Mio paragrafo </p>
```

Class

L'attributo **class** viene utilizzato per indicare un gruppo a cui l'elemento appartiene. È particolarmente utile quando si vuole far riferimento a più elementi in maniera concisa.

```
<p class="mio_paragrafo"> Questo  un mio paragrafo </p>
```

ID

L'attributo **id** viene utilizzato quando si vuole individuare univocamente un elemento all'interno della pagina.

```
<p id="my-id"> Questo  un mio paragrafo individuato univocamente </p>
```

Suggerimenti e best practices

Lo standard HTML non richiede l'utilizzo di caratteri minuscoli per i nomi degli attributi, ad esempio potremmo scrivere equivalentemente title="Info aggiuntive" o TITLE="Info aggiuntive". Tuttavia è raccomandato dal W3C il primo approccio, in quanto in linea con altri documenti che invece lo richiedono, come l'XHTML. Un'altra best practice è quella di racchiudere il valore degli attributi all'interno di virgolette o apici. Ad esempio potremmo scrivere href = miolink.xyz o href='miolink.xyz' al posto di href = "miolink.xyz". L'utilizzo degli apici al posto delle virgolette è necessario quando all'interno del valore sono presenti delle virgolette, come ad esempio title = 'Dwayne "The Rock" Johnson'. L'utilizzo di apici/virgolette è inoltre necessario se il valore contiene uno spazio al suo interno. È inoltre raccomandato che un id venga utilizzato per un unico elemento.

2.1.4 Tabelle

Le tabelle ci permettono di organizzare i dati all'interno di righe e colonne. Iniziamo con un esempio e analizziamo i vari elementi che appaiono:

```
<table>

    <tr>
      <th>Company</th>
      <th>Contact</th>
      <th>Country</th>
    </tr>

    <tr>
      <td>Alfreds Futterkiste</td>
      <td>Maria Anders</td>
      <td>Germany</td>
    </tr>

    <tr>
      <td>Centro comercial Moctezuma</td>
      <td>Francisco Chang</td>
      <td>Mexico</td>
    </tr>

</table>
```

- <**table**> è il tag che definisce la tabella, esso contiene al suo interno tutti gli altri elementi che costituiscono i vari blocchi.
- <**tr**> (table row) definisce una riga della tabella, contiene al suo interno tutti i blocchi relativi a tale riga.
- <**td**> (table data) definisce un blocco base della tabella.
- <**th**> (table header) definisce un blocco di intestazione della tabella.

La struttura di una tabella è abbastanza semplice, tuttavia questo è uno strumento molto potente in quanto permette una facile organizzazione dei dati all'interno della pagina. HTML ci mette inoltre a disposizione dei tag aggiuntivi che ci consentono di modellare più raffinatamente una tabella. Alcuni di questi tag includono:

- <**caption**> consente di aggiungere un titolo alla tabella: <caption> Mio titolo </caption>.
- <**colgroup**> e <**col**> definiscono rispettivamente un gruppo di colonne e le proprietà relative a quel gruppo di colonne.
- <**thead**> definisce l'header della tabella.
- <**tbody**> definisce il corpo della tabella.
- <**tfoot**> definisce il footer della tabella.

2.1.5 Liste

Le liste ci consentono di raggruppare elementi correlati tra loro sotto forma di liste. Le liste possono essere principalmente di due tipi: unordered e ordered. In entrambi i casi, gli elementi della lista sono specificati dal tag <li> (list item).

Unordered List

Una lista di questo tipo è specificata dal tag <ul> (unordered list), all'interno del quale sono specificati gli elementi.

```
<ul>
  <li>Caff</li>
  <li>T</li>
  <li>Latte</li>
</ul>
```

sarà visualizzata come segue

- Caffè
- Tè
- Latte

Orderd List

Una tale lista è specificata dal tag <ol> (orderd list). Gli elementi di questa lista sono associati a un numero, in base all'ordine in cui vengono dichiarati. Ad esempio, la stessa lista di sopra sarà visualizzata così:

1. Caffè
2. Tè
3. Latte

Description List

In aggiunta alle liste ordinate e non, HTML supporta le liste descrittive, definita dal tag <dl> (description list). A differenza dei due casi precedenti, gli elementi di questo tipo di lista sono composti da due tag: il primo è <dt> (data term) indica il nome dell'elemento, mentre il secondo è <dd> (data description) si occupa di descriverlo. Ad esempio:

```
<dl>
    <dt>Caff</dt>
    <dd>100% Arabica</dd>
    <dt>T</dt>
    <dd>Qualita' matcha verde</dd>
</dl>

<!--output-->
Caff
    - 100% Arabica
T
   - Qualita' matcha verde
```

2.2 CSS

Il CSS (Cascading Style Sheet) è la porzione di codice che si occupa di stilizzare i contenuti di una pagina web. Per applicare del CSS a una pagina html abbiamo due modi:

- Includere il codice nell'head della pagina all'interno di un tag <style>.
- Salvare il codice all'interno di un file apposito (con estensione .css) e poi importarlo nell'head della pagina tramite un tag link:

```
<link href="styles/style.css" rel="stylesheet">
```

Questo è l'approccio raccomandato, in quanto permette una più facile modularizzazione della pagina. L'attributo rel sta per relationship e indica che il file che stiamo importando avrà il ruolo di foglio di stile per la pagina.

Vediamo ora un esempio di codice CSS e analizziamone la struttura:

```
p {
    color: red;
}
```

Questo snippet rende rosso il colore del testo in tutti gli elementi paragrafo della pagina. Esso si divide in due macro parti: il selettore (in questo caso p) e il blocco delle dichiarazioni, ovvero le regole della forma proprietà:valore che si vogliono applicare (in questo caso color: red;). Le dichiarazioni devono essere racchiuse in un blocco tra parentesi graffe e separate tra loro da un punto e virgola.

2.2.1 Selettori

Ci sono vari tipi di selettori, ad esempio, la p nel codice di sopra è detto selettore di elemento, che seleziona tutti gli elementi di un tipo specifico, in questo caso tutti i paragrafi. Altri tipi di selettore includono:

- **Selettore di ID**: seleziona l'elemento con l'id specificato. Ad esempio

```
html:
<p id = "my-unique-id"> Questo testo  rosso </p>

css:
#my-unique-id {
    color: red;
}
```

- **Selettore di classe**: seleziona tutti gli appartenenti a una data classe, ad esempio

```
html:
<p class="my-class"> Questo testo rosso </p>
<h1 class="my-class"> Anche questo testo rosso </h1>

css:
.my-class {
    color: red;
}
```

- **Selettore di attributo**: seleziona gli elementi della pagina che hanno l'attributo specificato, ad esempio:

```
html:
<p title="info"> Questo paragrafo lo modifico </p>
<p> Questo no</p>

css:
p[title] {
    color: red;
}
```

- **Selettore di stato o pseudo-class selector**: selezionano un determinato elemento quando si trova in uno stato specifico. La sintassi è selettore_base:stato, dove il selettore base è uno qualunque tra quelli discussi fin'ora.
  Ad esempio, modifichiamo un paragrafo per rendere il testo rosso quando ci passiamo sopra col mouse:

```
html:
<p> Mio paragrafo </p>

css:
p:hover {
    color:red;
}
```

2.2.2 Dichiarazioni

In CSS, le dichiarazioni ci permettono di stilizzare un qualsiasi elemento in maniera molto approfondita. Vediamo un esempio per iniziare a prendere familiarità con alcune di queste:

```
body {
 width: 600px;
 margin: 0 auto;
 background-color: #ff9500;
 padding: 0 20px 20px 20px;
 border: 5px solid black;
}
```

Andiamo con ordine:

- Il selettore è **body**, ciò indica che le dichiarazioni riguardano tutti gli elementi della pagina html.
- **width:600px;** indica che il body avrà sempre larghezza 600px. Questo tipo di dichiarazione è assoluta e vedremo che non è un buona idea se si vogliono progettare dei siti responsive.
- **margin: 0 auto;** è una dichiarazione con più valori, il primo indica il margine verticale (che in questo caso è messo a zero), mentre il secondo indica il margine orizzontale (in questo caso, il valore auto divide lo spazio disponibile in maniera omogenea tra destra e sinistra). Questa sintassi è analoga per la proprietà **padding**.
- **background-color: #FF9500;** setta il colore dello sfondo tramite il codice hex passato. In generale, in css possiamo specificare un colore anche tramite tripla RGB o tramite il nome.
- **padding: 0 20px 20px 20px;** quando si usano quattro valori in una dichiarazione di questo tipo, essi fanno riferimento in ordine a sopra, destra, sotto e sinistra. Ciò è analogo per margin.
- **border: 5px solid black;** questa dichiarazione specifica larghezza, stile e colore del bordo di tutto il body.

Una descrizione di ogni possibile dichiarazione per ogni possibile elemento è fuori dal nostro focus per queste dispense, il mio consiglio è al solito di smanettare cercando sul web quando si hanno necessità specifiche. Per terminare il capitolo, concentriamoci su due concetti fondamentali: **position** e **flexbox**.

2.2.3 Position

Concentriamoci ora sulla proprietà **position**, che permette di specificare il tipo di posizionamento di un elemento. Questa può assumere in tutto 5 valori:

1. static
2. relative
3. fixed
4. absolute
5. sticky

Inoltre, settare la proprietà position abilita l'uso delle proprietà top, bottom, left e right, che consentono di posizionare effettivamente l'elemento. Il comportamento di queste ultime dipende inoltre dal valore dato a position!

position: static

Questo è il tipo di posizionamento di default per gli elementi html. In questo caso le proprietà top, bottom, left e right non hanno alcun effetto sull'elemento, il quale è sempre posizionato in accordo con la struttura html della pagina.

position: relative

Questo tipo di posizionamento fa si che le proprietà top, bottom, left e right modifichino la posizione dell'elemento rispetto alla sua posizione naturale. In questo caso la posizione degli altri contenuti non verrà aggiustata per colmare eventuali spazi vuoti lasciati dall'elemento.

position: fixed

Questo tipo di posizionamento fa sì che un elemento sia posizionato relativamente alla viewport, che significa che l'elemento starà sempre allo stesso posto anche quando la pagina viene scrollata. Ad esempio, se volessimo posizionare un elemento in basso a destra scriveremmo:

```
div.elemento {
     position: fixed;
     bottom: 0;
     right: 0;
     //altre eventuali dichiarazioni
}
```

position: absolute

Questo tipo di posizionamento fa sì che un elemento sia posizionato relativamente all'elemento che lo contiene che abbia anch'esso una dichiarazione position. Se un tale elemento non è presente questo posizionamento fa automaticamente riferimento al body del documento. Gli elementi posizionati in questo modo non seguono lo scorrere normale della pagina e possono sovrapporsi ad altri elementi.

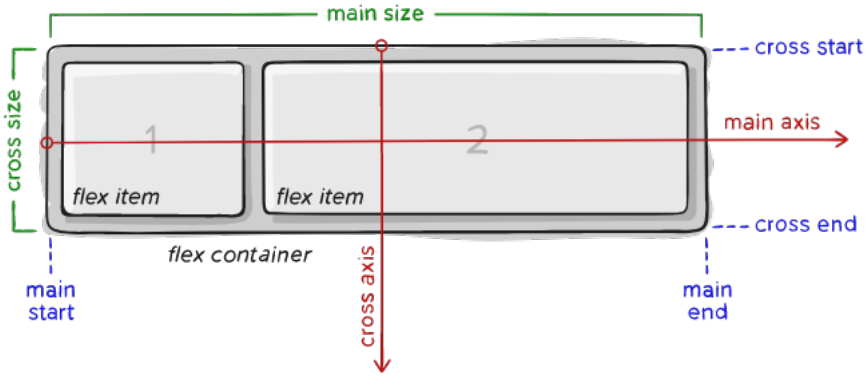
position: sticky

Un elemento con questo tipo di posizionamento è posizionato in base allo scroll in atto sull'elemento che lo contiene. In pratica un elemento sticky cambia il suo posizionamento tra relative e fixed in base alla posizione di scroll: è posizionato in maniera relative fino a che non si raggiunge una posizione di offset data, raggiunta tale posizione diventa fixed. Ad esempio, stilizziamo una div in modo che rimanga fissa in cima quando si raggiunge la sua posizione scrollando col mouse:

```
div.sticky {
     position: -webkit-sticky; /* Safari */
     position: sticky;
     top: 0;
 }
```

2.2.4 FlexBox

Il layout FlexBox (abbreviazione di Flexible Box) provvede una maniera efficiente di disporre e allineare elementi all'interno di un contenitore, anche quando le loro dimensioni sono sconosciute e/o dinamiche. L'idea principale che sta dietro a quest'approccio è dare all'elemento contenitore il potere di modificare ordine e dimensioni degli elementi al suo interno in maniera da gestire in modo ottimale lo spazio disponibile in base al display del dispositivo utilizzato. Un flex container si espande e riduce automaticamente a sua volta a tale scopo.
Iniziamo a vedere la terminologia e le proprietà associate a questo layout. Iniziamo con dare un'occhiata alla seguente immagine per capire il flow degli elementi:



- **main axis**: è l'asse principale rispetto a cui gli elementi del container sono posizionati. Questa non è necessariamente orizzontale, ma dipende dalla proprietà **flex-direction** (che tratteremo tra poco).
- **main-start | main-end**: indicano le estremità rispetto a cui gli elementi sono posizionati.
- **main size**: corrisponde alla dimensione principale tra l'altezza e la larghezza.
- **cross axis**: è l'asse perpendicolare a quella principale. La sua direzione dipende dalla direzione di quest'ultimo.
- **cross-start | cross-end** Indicano la direzione delle linee su cui vengono disposti gli oggetti.
- **cross size** è la dimensione secondaria rispetto alla main size.

Vediamo ora le proprietà legate a flexbox, iniziando da quelle del container:

Proprietà del flex container

- **display: flex** definisce l'elemento come un flex container, abilitando quindi tutte le proprietà flex per gli elementi al suo interno.
- **flex-direction**: stabilisce la direzione e il verso del main-axis. I valori che può prendere sono:
  1. row: (valore di default) da sinistra a destra.
  2. row-reverse: da destra a sinistra.
  3. column: dall'alto verso il basso.
  4. column-reverse: dal basso verso l'alto.
- **flex-wrap**: indica come organizzare gli item nelle varie righe. I valori che può prendere sono:
  1. nowrap: (valore di default) posiziona tutti gli elementi in un'unica riga.
  2. wrap: dispone gli elementi su più righe, in ordine dall'alto verso il basso.
  3. wrap-reverse: uguale a wrap ma dal basso verso l'alto.
- **flex-flow**: combina le due precedenti e ha come valore di default row nowrap (i.e. i due valori di default rispettivi). Accetta come valore una qualsiasi combinazione dei valori di sopra.
- **justify-content**: definisce la disposizione dei contenuti rispetto al main axis. I valori che può prendere sono:
  1. flex-start: (default) gli elementi sono spinti verso l'inizio della flex-direction.
  2. flex-end: come sopra ma verso la fine.
  3. start (/ end): gli elementi sono spinti verso l'inizio (/fine) della direzione indicata in writing-mode.
  4. left (/ right): gli elementi sono spinti verso sinistra (/destra).
  5. center: gli elementi sono disposti al centro della riga a cui appartengono.
  6. space-between: gli elementi sono distribuiti uniformemente sulla riga, col primo attaccato all'inizio e l'ultimo attaccato alla fine.
  7. space-around: gli elementi sono distribuiti uniformemente sulla riga.
  8. space-evenly: gli elementi sono disposti in modo che lo spazio tra due elementi 'confinanti' sia sempre lo stesso.
- **align-items**: definisce come gli elementi sono disposti sulla loro riga rispetto al cross-axis. I valori che può prendere sono:
  1. stretch: (default) adatta le dimensioni degli elementi per coprire lo spazio disponibile.
  2. flex-start / start / self-start: gli elementi sono disposti all'inizio del cross-axis. Le differenze tra questi sono analoghe al caso del justify-content.
  3. flex-end / end / self-end: come sopra ma rispetto alla fine del cross-axis.
  4. center
  5. baseline: gli elementi sono allineati in modo che le loro basi siano sullo stesso livello.
- **align-content**: è utilizzato per disporre le righe di un flex container quando vi è dello spazio in più nella direzione del cross-axis. I valori che può prendere sono:
  1. normal: (default)
  2. flex-start / start: le righe sono organizzate all'inizio del container. Flex-start segue la flex-direction mentre start la direzione di writing-mode.
  3. flex-end/end: come sopra ma rispetto alla fine del container.
  4. center
  5. space-between: le righe sono distribuite uniformemente, la prima all'inizio del container e l'ultima alla fine.
  6. space-around / space-evenly / stretch: analoghe al caso di justify content ma per le righe.

Proprietà dei flex item

- **order**: controlla l'ordine in cui l'elemento appare nel container.
- **flex-grow**: indica l'abilità di un elemento di aumentare le proprie dimensioni se necessario. Prende come valore un numero puro (senza unità di misura) che serve come proporzione. Se tutti gli elementi hanno questa proprietà con valore 1, lo spazio rimanente viene distribuito in maniera uniforme tra essi; se uno di questi la ha a 2 riceverà il doppio dello spazio rispetto agli altri e così via.
- **flex-shrink**: come sopra, ma riguarda l'abilità di decrescita.
- **flex-basis**: definisce le dimensioni di partenza prima che lo spazio rimanente venga distribuito. Il valore può essere una lunghezza o una keyword:
  1. auto: rimanda alla dimensione principale.
  2. content: rimanda alla dimensione del contenuto dell'elemento.
- **flex**: combina flex-grow, flex-shrink e flex-basis, con il secondo e terzo parametro opzionale e valori di default 0 1 auto. È raccomandata rispetto a usare le tre proprietà individuali per brevità.
- **align-self**: sovrascrive il valore della proprietà align-items del container per l'elemento in questione. Può prendere gli stessi valori di quest'ultima.

A questo link trovate un comodo cheat-sheet per la visualizzazione di queste proprietà.

Capitolo 3

Angular

Angular è una piattaforma di sviluppo basata su TypeScript. Tra le sue features, angular include:

- Un framework componen-based per lo sviluppo di single page web-app scalabili.
- Un insieme di librerie built-in che provvedono una vasta varietà di funzionalità, tra cui supporto alla navigazione, gestione dei form e comunicazione client-server.
- Una developer tools suite che copre le fasi di sviluppo, testing, build e update dell'applicazione.

Alcuni dei vantaggi e svantaggi comportati dall'utilizzo di angular includono:

Vantaggi

- Implementazione dell'architettura MVVM (Model View View-Model), che consente di isolare la logica dal layer di interfaccia grafica e quindi di separare effettivamente le funzionalità dell'app, rendendo più facile lo sviluppo e il mantenimento. In questa architettura, il layer Model è rappresentato dai DTO che codificano gli oggetti con cui l'app avrà a che fare (che, per garantire una corretta implementazioen di un'app full stack, devono essere uguali ai DTO definiti nel lato server). Il layer View è invece rappresentato dai file .html e .css, che curano appunto la parte grafica della nostra app. Il layer View-Model è infine dato dai file TypeScript che regolano la dinamicità delle componenti, rappresentando di fatto la logica di collegamento tra la View e il Model.
- Struttura dell'app in moduli, che consente di incapsulare varie funzionalità strettamente legate tra loro all'interno di un unico scope.
- Utilizzo dei service e implementazione della Dependency Injection, con tutti i vantaggi che questo comporta (vedere discussione qui **??**).
- Utilizzo di TypeScript come linguaggio, che consente di eliminare gran parte degli errori più comuni in fase di sviluppo, rendendo il codice più pulito e facile da comprendere rispetto a JavaScript.

Svantaggi

- Curva di apprendimento molto ripida, in particolare se paragonata ad altri framework quali REACT o VUE.
- Verbosità e complessità.
- Limitatezza nelle opzioni delle SEO (Search engine optimization) e una poco supportata accessibilità ai search engine crawlers (che di base consentono la calssificazione dei risultati di una ricerca in ordine di rilevanza).

Per terminare l'introduzione, descriviamo brevemente come installare Angular e creare un primo progetto vuoto. Per prima cosa, abbiamo bisogno di installare la command line interface (CLI) di Angular. Per fare ciò possiamo affidarci a npm (Node Package Manager), che ci viene fornito nell'installazione base di NodeJS (un potente runtime environment per JS).
Dopo aver installato Node apriamo un terminale e lanciamo il comando npm install -g @angular/cli (il flag -g sta per global e rende anhular disponibile all'intero sistema). Terminata l'installazione, possiamo creare la struttura base di un progetto direttamente sa linea di comando tramite ng new **nome-app**. Questo comando genera in automatico tutte le configurazioni di base e inizializza il modulo principale, inoltre ci consente qualora lo volessimo (ovvero sempre) di aggiungere le opzioni di routing alla nostra app. Non male per un singolo comando!
Possiamo già lanciare la nostra applicazione base: spostiamoci all'interno della cartella del progetto (che verrà creata automaticamente dal comando precedente) e lanciamo l'app sempre dalla cli tramite ng serve -o (Dove il flag -o oppure l'equivalente –open aprono automaticamente il broswer al localhost:4200, che è la porta di default per le app angular). Terminate le fasi iniziali, addentriamoci nel funzionamento del framework!

3.1 Moduli

Un modulo, come anticipato in precedenza, consiste nella cellula base di un progetto Angular, all'interno del quale vengono incluse tutte le funzionalità che sono legate strettamente tra loro. Alla creazione del progetto, angular ci fornisce in partenza l'app-module. Questo è il modulo principale dell'app, composto da 5 file:

- app.module.ts: è il file che gestisce di fatto l'applicazione: al suo interno vengono dichiarati tutti gli altri moduli (sia interni che importati) che verranno utilizzati nell'applicazione. Viene inoltre specificata la componente da bootstrappare all'avvio dell'applicazione.
- app.component.ts /.html /.css: sono i tre file che definiscono la componente principale associata a questo modulo. Esploreremo più nel dettaglio le componenti nella prossima sezione.
- app-routing.module.ts: è il file in cui vengono specificate le path, ovvero delle coppie url/component che verranno utilizzate dal router per la navigazione all'interno dell'applicazione.

In generale, per creare un nuovo modulo utilizzeremo il commando ng generate module **nome-modulo**. Lanciando questo comando verrà in automatico creata una nuova directory, all'interno della quale potremo un file nome.module.ts. Se vogliamo aggiungere un file per il routing interno al modulo possiamo runnare lo stesso comando aggiungendo il flag –routing prima del nome. Esaminiamo la struttura di un file *.module.ts: la prima cosa che notiamo è il decoratore @NgModule. Questo definisce che un tale file contiene una classe che rappresenta un modulo angular. Questo decoratore prende come argomento un oggetto JSON che viene utilizzato per specificare varie funzionalità del modulo:

1. declarations: specifica le componenti contenute all'interno del modulo. Di default è vuoto (in quanto non abbiamo ancora creato nessuna componente).
2. imports: specifica i moduli che vengono importati all'interno del modulo. Gli exportables (ovvero gli oggetti dichiarati con la keyword **export** presenti in questi moduli saranno utilizzati all'interno di ogni template del nostro modulo.) Di default contiene CommonModule, il quale fornisce le funzionalità di base di Angular che approfondiremo in seguito, e (se lo abbiamo aggiunto) il relativo modulo di routing. Solamente queste prime due proprietà (declarations e imports) saranno presenti alla creazione del modulo.
3. providers: nel quale vengono specificati tutti i moduli importati dei quali si vuole utilizzare un injectable, ovvero un oggetto di cui si vuole iniettare la dipendenza all'interno di una componente del nostro modulo.

4. bootstrap: la lista delle componenti che vengono automaticamente lanciate quando il nostro modulo viene lanciato.

La classe decorata da @NgModule è generalmente vuota e dichiarata tramite la keyword **export** se si vuole rendere disponibile il contenuto del modulo all'interno dell'app. Esaminiamo come queste proprietà vengano utilizzate nel routing module: all'interno di questo file viene innanzitutto dichiarata una const di tipo Routes (ovvero un array di oggetti Route), la quale specifica una lista di coppie url/component La struttura di tale array è la seguente:

```
const routes: Routes = [
     { path: "/miopath", component: MiaComponent }
];
```

Un oggetto di tipo route comprende inoltre la proprietà opzionale **children**, la quale prende come valore un array di Route, i path delle quali sono da considerarsi relativi al path della route madre. Ad esempio:

```
const routes: Routes = [
     { path: "/miopath", component: MiaComponent, children: [
             {path: "/figlio1", component: PrimoFiglio},
             {path: "/figlio2", component: SecondoFiglio}
         ]
     }
];
```

Questa const verrà poi passata come argomento al metodo forRoot del RouterModule all'interno degli import, il quale crea e configura un modulo con tutte le routes e le direttive fornite. Alla fine la struttura sarà qualcosa di simile:

```
const routes: Routes = [
     { path: "/miopath", component: MiaComponent, children: [
             {path: "/figlio1", component: PrimoFiglio},
             {path: "/figlio2", component: SecondoFiglio}
         ]
     }
];

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

3.2 Componenti

Le compoonenti sono il building block principale di una applicazione angular. Ciascuna di esse consiste di tre file:

1. un template HTML che specifica ciò che viene mostrato a schermo.
2. una classe TypeScript che regola il comportamento della componente.
3. Un file CSS (opzionale) che definisce lo stile della componente.

Per creare una componente, apriamo un terminale all'interno del modulo a cui vogliamo che questa componente appartenga e lanciamo il comando ng generate component **nome-comp** dalla cli. Questo creerà una cartella contenente i tre file menzionati qui sopra, assieme a un file contente le specifiche per il testing. Inoltre registrerà automaticamente la component all'interno delle declarations del modulo contenitore.
Esaminiamo in particolare il file .component.ts, nel quale è presente la classe che gestisce il comportamento della componente. Tale classe è decorata con @Component, il quale specifica al suo interno tre proprietà:

1. selector: specifica il selettore css che può essere utilizzato nei template per includere la componente come una direttiva. Di default ha la forma app-nome-comp.
2. templateUrl: specifica il path relativo o l'url del file html della component. Di default fa riferimento al file creato automaticamente dal comando.
3. styleUrls: definisce un array di url relativi ai file css da utilizzare per lo stile della componente. Di default l'array include solamente il file creato automaticamente dal comando.

All'interno della classe (dichiarata come exportable) troviamo invece tutta la logica che regola il comportamento della view. Di default essa implementa l'interfaccia OnInit, la quale espone un metodo (ngOnInit(): void) che consente di specificare tutte le azioni da effettuare ogni volta che la componente viene lanciata. Contiene inoltre un costruttore, il quale verrà utilizzato principalmente per andare a iniettare le dipendenze che ci occorrono (vedremo più nel dettaglio come in seguito).
Ad esempio, la seguente componente contiene una variabile che viene inizializzata appena la classe viene creata:

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

@Component({
  selector: 'app-superdummy',
  templateUrl: './superdummy.component.html',
  styleUrls: ['./superdummy.component.css']
})
export class superdummyComponent implements OnInit {

  miaVar: string;

  constructor() { }

  ngOnInit(): void {
    miaVar = "Sono inizializzata";
  }

}
```

Il metodo ngOnInit è detto lifecycle hooks, ovvero un metodo che regola un momento della vita della componente, in questo caso la creazione. Esso è l'unico di questi eventi che si deve obbligatoriamente implementare dal contratto con l'interfaccia, i restanti sono comunque utili e una loro descrizione dettagliata è consultabile qui.

3.2.1 Comunicazione tra componenti

Sarebbe conveniente avere un modo per condividere i dati tra una componente madre e le rispettive componenti figlie. Angular ci viene incontro in ciò tramite l'utilizzo di due decoratori: @Input() e @Output(), entrambi importabili da @angular/core. @Input() permette alla component madre di aggiornare dei dati nella component figlia, @Output() permette l'analogo da figlia a madre. Vediamo come utilizzare entrambi:

@Input()

Per utilizzare questo decoratore è necessario configurare sia la componente madre che quella figlia. Supponiamo ad esempio di voler passare una stringa, innanzitutto dichiariamo la variabile nella component figlia e decoriamola:

```
@Component({
 selector: 'app-figlia',
 templateUrl: './figlia.component.html',
 styleUrls: ['./figlia.component.css']
 })
 export class figliaComponent implements OnInit {

 @Input() myVar: string = '';

 constructor() { }

 ngOnInit(): void {
 }

 }
```

In seguito leghiamo tale stringa a un dato presente nella component madre come segue:

1. richiamiamo la componente figlia nel template della madre tramite il suo selettore CSS.
2. utilizzaimo il data binding per legare myVar alla variabile che vogliamo passare all'interno del tag (ad esempio alla variabile "varMadre" opportunamente dichiarata nella component madre)

```
<app-figlia
    [myVar]="varMadre"
></app-figlia>
```

@Output()

Questo decoratore funziona in maniera leggermente differente: esso decora una proprietà nella component figlia che si vuole trasmettere alla component madre. Vediamo come:

1. si codifica tale proprietà come un EventEmitter (importabile anche esso da @angular/core) e lo si decora con @Output(). Supponiamo ad esempio di voler comunicare una stringa:

```
@Component({
 selector: 'app-figlia',
 templateUrl: './figlia.component.html',
 styleUrls: ['./figlia.component.css']
 })
 export class figliaComponent implements OnInit {

     @Output() myVar = new EventEmitter<string>();

     constructor() { }

     ngOnInit(): void {
     }

 }
```

2. Sempre all'interno della classe figlia si crea un metodo che emetta un evento con un valore specifico. Tale valore è quello che ci interessa propagare alla component madre. Tale metodo può essere triggerato ad esempio da un pulsante:

```
export class figliaComponent implements OnInit {

  @Output() myVar = new EventEmitter<string>();

  constructor() { }

  ngOnInit(): void {
  }

  public emetti(value: string) {
      this.myVar.emit(value);
  }

}
```

3. Nella componente madre, si crea un setter per la variabile che vogliamo ricevere dalla figlia:

```
varMadre: string;

public setVarMadre(value: string) {
    this.varMadre = value;
}
```

4. Infine, si lega l'emitter della componente figlia al metodo creato in precedenza. Anche in questo caso, questo viene fatto all'interno del tag CSS che specifica la componente figlia nel template della madre.

```
<app-figlia
    (myVar) = "setVarMadre($event)"
></app-figlia>
```

3.3 Template e Data Binding

Angular estende le funzionalità base dell'HTML, consentendo la realizzazione di un DOM (document object model) dinamico. Ricordiamo che Angular consente la realizzazione di single page applications, di conseguenza la struttura base html (<html>, <head> e <body>) si definisce solamente all'interno dell'index e non deve essere ripetuta nei template delle componenti. Inoltre i template angular non supportano emlementi <script>, per ridurre il rischio sùdi attcchi script-injection.

3.3.1 Interpolazione

L'interpolazione ci consente di visualizzare un dato definito nel TS della component all'interno del template. Questo dato deve essere delimitato da una coppia di doppie parentesi graffe. Ad esempio, se vogliamo mostrare a schermo il valore di una variabile scriveremmo

```
//nel TS
customer: CustomerDto; //al suo interno ho una property 'nome'
/* inizializzo il customer in qualche modo*/

//nell'HTML
<p> Nome del cliente: {{customer.nome}}</p>
```

L'interpolazione può essere usata anche all'interno dei tag, ad esempio per settare la fonte di un'immagine:

```
//nel TS
myUrl: string = "sitosicuro.it/immaginesicuratrust";

//nell'HTML
<img src="{{myUrl}}" ... >
```

3.3.2 Data Binding

Angular fornisce tre tipi di binding, in base alla direzione del legame:

1. One-Way dalla source alla view (dal TS all'HTML): utilizza una coppia di parentesi quadre e la sintassi [target]=”espressione” (anche l'interpolazione fa parte di questa categoria).
2. One-Way dalla view alla source: utilizza una coppia di parentesi tonde e la sintassi (target) = ”statement”.
3. Two-Way che lega i dati dalla source alla view e viceversa. Utilizza una coppia di parentsi ”a banana” e la sintassi [(target)] = ”espressione”.

Il target di un data binding può essere una proprietà, un evento o anche il nome di un attributo. Tramite data binding ci si può legare a un qualunque membro del TS che sia pubblico.
Esaminiamo ora i diversi tipi di binding, assieme ai target a cui fanno riferimento.

- **Property Binding**: può avere come target una proprietà di un elemento, di una component o di una direttiva. Nel seguente esempio, facciamo property binding sulle proprietà alt, src e ngClass:

```
<img [alt]="hero.name" [src]="heroImageUrl">

<div [ngClass]="{'special': isSpecial}"></div>
```

- **Event Binding**: può avere come target un evento di un elemento, di una component o di una direttiva. Nel seguente esempio, facciamo event binding su click e myClick:

```
<button type="button" (click)="onSave()">Save</button>

<div (myClick)="clicked=$event" clickable>click me</div>
```

- **Two-way binding**: può avere come target sia eventi che proprietà, ad esempio

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

3.4 Direttive

Le direttive Angular sono classi che provvedono dei comportamenti aggiuntivi agli elementi di una applicazione. Queste si dividono in due tipi:

- **Direttive d'attributo**: modificano l'aspetto e il comportamento di un elemento, di una componente o di un'altra direttiva.
- **Direttive strutturali**: cambiano il layout del DOM (document object model), aggiungendo o rimuovendo elementi del layout.

Direttive d'attributo

La più comuni direttive d'attributo built in sono **NgClass**, **NgStyle** e **NgModel**.
NgClass aggiunge e rimuove un insieme di classi CSS allo stesso tempo. Essa può essere usata con espressioni ternarie o metodi appositi. Ad esempio:

```
    //se isSpecial = true viene assegnata la classe special alla div

<div [ngClass]="isSpecial ? 'special' : ''">...</div>
```

Per un esempio dell'utilizzo con un metodo guardate qui.
NgStyles viene utilizzato per impostare più proprietà css di un elemento in maniera inline (ovvero all'interno del tag). Il suo utilizzo è utile per cambiare dinamicamente lo stile in base a una o più proprietà. Per utilizzarlo, ci si serve di un metodo dichiarato nella componente, all'interno del quale si setta un oggetto corrispondente alle proprietà CSS che si vogliono assegnare.

```
    currentStyles: Record<string, string> = {};
    /* dichiaro tre booleani canSave, isUnchanged e isSpecial e li inizializzo
        a true*/
    setCurrentStyles() {
      // CSS styles: set per current state of component properties
      this.currentStyles = {
        'font-style':  this.canSave      ? 'italic' : 'normal',
        'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
        'font-size':   this.isSpecial    ? '24px'   : '12px'
      };
    }
```

Infine, si lega tale metodo alla proprietà ngStyle nel tag dell'elemento desiderato:

```
<div [ngStyle]="currentStyles">
     Questa div ha di default font-style italic, font-weight bold e font-
         size 24px.
</div>
```

NgModel viene utilizzato per mostrare una proprietà definita nella logica e per aggiornare tale proprietà quando l'utente opera delle modifiche. È utilizzata in maniera particolare all'interno dei form per legare un input a una variabile definita nel typescript corrispondente. Ad esempio, supponiamo di aver definito una variabile nomeAttuale nel typescript e di volerla legare a un input di un form:

```
    <input type="text" [(ngModel)]="nomeAttuale">
```

In questo modo il contenuto di tale variabile sarà corrispondente al contenuto di tale elemento.

Direttive strutturali

Le direttive strutturali sono responsabili del layout HTML dell'applicazione. Queste possono aggiungere o rimuovere gli elementi a cui fanno riferimento. Le più comuni sono **NgIf**, **NgFor** e **NgSwitch**.
NgIf aggiunge o rimuove un elemento in base al valore di un booleano: se tale valore è true l'elemento (e tutto ciò che contiene) viene mostrato, se è false no. Per utilizzare questa direttiva basta dichiararla nel tag dell'elemento tramite la sintassi *ngIf="myBool", dove myBool è il booleano a cui si vuol fare riferimento. Questa direttiva supporta anche una forma if-else, con la seguente sintassi:

```
<div *ngIf="myBool; else tagDellElse">
     Contenuto mostrato se myBool=true
 </div>
<ng-template #tagDellElse>
      Contenuto mostrato se myBool=false
</ng-template>
```

ngFor viene utilizzato per mostrare una lista di elementi dinamicamente. Supponiamo ad esempio di aver definito un array di stringhe items nel TS, per mostrarli tutti a schermo nel template si utilizza *ngFor nel tag dell'elemento che si vuole ripetere. Ad esempio:

```
<div *ngFor="let item of items"> {{item.name}} </div>
```

NgFor supporta anche la sintassi estesa *ngFor="let item of items; let i=index", la quale ci consente anceh di avere a disposizione l'indice di ogni elemento che viene ripetuto. Ciò è particolarmente utile quando si vuole far riferimento a un particolare elemento dell'iterazione.
NgSwitch sceglie che 'forma' di un elemento mostrare in base al valore di una determinata variabile interruttore. Esso imita la sintassi di uno switch Java/Javascript. Se ad esempio vogliamo cambiare cosa viene mostrato all'interno di una div in base a una variabile orarioAttuale faremmo così:

```
<div [ngSwitch]="orarioAttuale">
    <p *ngSwitchCase="mattina"> Buongiornissimo! </p>
    <p *ngSwitchCase="pomeriggio"> Buon pomeriggio! </p>
    <p *ngSwitchCase="sera"> Buonaseeeeeeeee' </p>
    //se orarioAttuale non ha nessuno dei tre valori precedenti mostro
        questo
    <p *ngSwitchDefault> WEWE AMICI DEL WEB</p>
</div>
```

3.5 Dependency Injection

La dependency injection è uno dei meccanismi fondamentali di angular come framework. Questo meccanismo permette alle classi decorate coi relativi decoratori di configurare autonomamente le dipendenze a loro necessarie. In questo meccanismo, chi fruisce della dipendenza è detto consumer mentre chi la fornisce provider. Angular facilita l'interazione tra queste due parti grazie a un'astrazione detta injector. Quando una dipendenza viene richiesta dal consumer, l'injector controlla se un'istanza è già presente, altrimenti ne crea una nuova e la salva nel registro. Un injector globale (detto root injector) è creato da Angular all'avvio dell'applicazione. Vediamo ora nel dettaglio come si creano i providers e come si iniettano nei consumers.

Provider

Affinché una classe possa essere iniettata all'interno di un'altra, essa deve essere decorata con @Injectable().

```
@Injectable()
export class MyService { ... }
```

In seguito, dobbiamo rendere disponibile tale dipendenza all'interno dell'app. Una dipendenza può essere resa disponibile in tre livelli differenti:

- Component-Level: dichiarando la classe nell'array dei providers. Ad esempio:

```
@Component({
    selector: 'my-comp',
    template: '...',
    providers: [MyService]
})
```

  In questo caso MyService (e tutte le sue funzionalità) sono disponibili all'interno di MyComponent e di tutte le componenti e direttive che vengono utilizzate nel template. Quando si registra un provider a questo livello, si ottiene una nuova istanza ogni volta per ogni istanza della componente.

- NgModule-Level: dichiarando la classe nell'array dei providers analogamente a sopra. In questo caso, una stessa istanza della classe è disponibile per tutte le componenti del modulo.

- Root-Level: questo è fattibile in due modi. O dichiarando la classe nei providers del'app-module oppure all'interno del decoratore @Injectable(), utilizzando la proprietà providedIn:

```
@Injectable({
    providedIn: 'root'
})
export class MyService { ... }
```

  In questo caso, una singola istanza del service è disponibile in tutta l'applicazione. Questo consente inoltre ad angular di rimuovere il servizio dalla app compilata qualora questo non fosse utilizzato, ottimizzand così le prestazioni.

Consumers

Il modo più comune iniettare una dipendenza all'interno di una classe è utilzzare il costruttore. Quando Angular crea una nuova istanza di una classe, determina di quali dipendenze questa abbia bisogno guardando i tipi dei parametri dichiarati nel costruttore. Una volta che trova le dipendenze necessarie (e eventualmente istanzia quelle mancanti) angular passa tali istanze al costruttore per creare effettivamente la classe. Ad esempio, supponiamo di voler iniettare MyService all'interno di una component:

```
@Component({
    selector: 'my-comp',
    template: '...',
    providers: [MyService]
})
export class MyComponent implements OnInit {

    constructor(private service: MyService) {}
```

```
        ngOnInit(): void {
        }

    }
```

3.6 Collegamento col Back-End

Concentriamoci ora nello specifico sull'utilizzo di Angular per lo sviluppo del lato client all'interno di un'architettura REST. Per fare ciò si utilizzano dei service appositi, i quali possono essere iniettati nelle component ogniqualvolta si voglia fare una comunicazione con il back-end.
Prima di entrare nel dettaglio su come questa comunicazione venga effettuata, abbiamo bisogno di approfondire un concetto fondamentale di Angular: gli observable.

3.6.1 Observable

Gli observable svolgono un ruolo di supporto nella comunicazione di più parti di un'applicazione. Essi seguono il design pattern dell'observer in cui un oggetto detto subject gestisce una lista di suoi dipendenti, chiamati observers e comunica ad essi ogni cambio di stato automaticamente. Gli Observable sono dichiarativi, ovvero essi non emettono i valori gestiti fino a che non vengono chiamati da un utente, tramite una procedura detta subscribe. Vediamo quindi come vengono implementati praticamente: per creare un observable in TS si crea un'istanza della classe Observable<T>, dove T è il tipo degli oggetti che si vogliono gestire tramite quell'observable. Questa classe espone il metodo .subscribe(), che viene utilizzato quando si vuole accedere agli oggetti gestiti e permette di specificare come ottenere e generare valori o messaggi dall'observable. Il metodo subscribe prende come argomento un observer: questo è un oggetto TS che definisce la gestione delle notifiche inviate dall'observable. Il metodo subscribe restituisce un oggetto di tipo Subscription, il quale contiene un metodo .unsubscribe(), che può essere utilizzato per terminare la ricezione di notifiche dall'observable.
I tipi di notifica che un observable può inviare sono tre e e vengono gestiti da tre handler nell'observer che si passa al metodo subscribe. Questi handler sono esplicitati come function e sono i seguenti:

1. **next**: è l'unico handler obbligatorio, che definisce come gestire ciascun valore emesso.
2. **error**: è l'handler che specifica il comportamento da seguire in caso di errore. Un errore termina l'esecuzione dell'observable.
3. **complete**: specifica che notifica inviare al termine dell'esecuzione dell'observable.

Gli ultimi due handler sono opzionali: se non vengono specificati un observable semplicemente non invierà alcuna notifica in caso di errore o al termine dell'esecuzione. Vediamo ora un esempio pratico, in cui creiamo artificialmente un observable tramite il metodo of(... items) della libreria RxJS, il quale restituisce un Observable che gestisce i valori passati.

```
const myObservable = of(1,2,3);

const myObserver = {
     next: (x: number) => console.log("Valore attuale: " + x),
     error: (err: Error) => console.log("Errore inaspettato, termino l'
         esecuzione"),
     complete: () => console.log("Esecuzione terminata"),
};
```

```
//facciamo il subscribe passando myObserver: questo poteva essere anche
    dichiarato esplicitamente direttamente nel subscribe
myObservable.subscribe(myObserver);

// Logs:
// Valore attuale: 1
// Valore attuale: 2
// Valore attuale: 3
// Esecuzione terminata
```

La funzione subscribe supporta inoltre la dichiarazione inline delle function:

```
//forma con observer
myObservable.subscribe({
     next: (x: number) => console.log("Valore attuale: " + x),
     error: (err: Error) => console.log("Errore inaspettato, termino l'
         esecuzione"),
     complete: () => console.log("Esecuzione terminata"),
});

//forma equivalente con handler inline
myObservable.subscribe(
     x => console.log("Valore attuale: " + x),
     err => console.error("Errore inaspettato, termino l'esecuzione"),
     () => console.log("Esecuzione terminata")
);
```

3.6.2 Utilizzo delle chiamate HTTP

Ora che abbiamo un'idea di cosa siano gli observable e di come funzionino, vediamo come vengono utilizzati per gestire le comunicazioni col back-end. Prima di entrare nel vivo della discussione, parliamo un attimo degli environments.
Quando creiamo un progetto da cli, Angulare crea automaticamente la cartella environments, con al suo interno due file TS: environment.ts e environment.prod.ts. Questi file hanno lo scopo di contenere le variabili d'ambiente da utilizzare rispettivamente nelle fai di development e di produzione della nostra applicazione. Angular sostituisce automaticamente il secondo al primo in fase di produzione, grazie alla dichiarazione nell'array fileReplacements, all'interno del file di configurazione angular.json. Nel nostro caso, utilizzeremo questi due file per salvare gli url del back-end sia in fase di development (tipicamente la porta locale 8080 se lavoriamo con Spring) che in fase di produzione.

```
//contenuto del file environment.ts

export const environment = {
    production: false,      //presente di default alla creazione
    apiEndPoint:"http://localhost:8080/"    //variabile aggiunta da noi
};
```

Questi URL ci serviranno quando andremo a fare le chiamate vere e proprie nei service. Vediamo innanzitutto un esempio di un service base e analizziamolo passo passo. Per prima cosa, creiamo una cartella service e apriamola nel terminale. Lanciamo quindi il comando ng generate service nome-service, questo creerà automaticamente due file: nome-service.service.ts e un file con le specifiche di testing, nome-service.service.spec.ts. Il service sarà già configurato come injectable globale.

```
import { Injectable } from '@angular/core';
import { environment } from 'src/environments/environment';

@Injectable({
  providedIn: 'root'
})
export class DummyService {

  constructor() { }

}
```

Per prima cosa importiamo la const evironment, che utilizzeremo per recuperare l'endpoint dell'API del back-end. In seguito, iniettiamo un HttpClient nel costruttore. Per fare ciò, dobbiamo importare il modulo HttpClientModule nel modulo principale dell'applicazione e dichiararlo nei provider sempre all'interno di questo. Questo oggetto espone al suo interno i vari metodi utilizzati per effettuare le varie chiamate HTTP, semplificandoci notevolmente la vita.

```
import { Injectable } from '@angular/core';
import { environment } from 'src/environments/environment';
import { HttpClient } from '@angular/common/http';
@Injectable({
  providedIn: 'root'
})
export class DummyService {

  constructor(private http: HttpClient) { }

}
```

Infine, definiamo i vari metodi che si occuperanno delle chiamate. Al loro internoutilizzeremo i metodi forniti dal HttpClient, i quali restituiscono **un Observable** che gestisce dati del tipo restituito dal metodo del backend che si invoca (qualora non si fosse sicuri si può anche passare any come argomento generico dell'Observable). Vediamo come esempio cinque metodi che utilizzano le cinque chiamate principali http, operando con un generico data transfer object.

```
import { Injectable } from '@angular/core';
import { environment } from 'src/environments/environment';
import { HttpClient } from '@angular/common/http';
import { DTO } from 'src/dtos/dto'; //questo e' il file che contiene la
    classe DTO al suo interno

@Injectable({
  providedIn: 'root'
})
export class DummyService {

  constructor(private http: HttpClient) { }

  public getAll(): Observable<DTO[]> {
    return this.http.get<DTO[]>(environment.apiEndPoint + "getAll");    //
        prende come argomento l'url del back che si vuole contattare
  }

  public delete(id: number): Observable<any> {
```

```
    return this.http.delete(environment.apiEndPoint + "delete?id=" + id);
        //in questo caso l'id viene passato come parametro della request
        direttamente nell'url
  }

  public insert(dto: DTO): Observable<DTO> {
    return this.http.post<DTO>(environment.apiEndPoint + "insert",
                          dto); //il secondo param corrisponde al body
                              della request
  }

  public update(dto: DTO): Observable<DTO> {
    return this.http.put<DTO>(environment.apiEndPoint + "update",
                          dto);
  }

  public patch(patchDTO: JSON): Observable<DTO> {
    return this.http.patch<DTO>(environment.APIEndpoint + "patch",
                                patchDTO);
  }


}
```

Poichè restituiamo un Obserable, quando chaimeremo questi metodi all'interno di una component dovremmo utilizzare il metodo subscribe per poter accedere ai risultati. Questo è concatenabile in maniera funzionale alla chiamata del metodo nel service. Vediamo un esempio, in cui utilizziamo il metodo getAll per salvare tutte le entità di un database all'interno di un array.

```
import { Component, OnInit } from '@angular/core';
import { DTO } from 'src/dtos/dto';
import { DummyService } from 'src/service/dummy.service';

@Component({
  selector: 'app-superdummy',
  templateUrl: './superdummy.component.html',
  styleUrls: ['./superdummy.component.css']
})
export class superdummyComponent implements OnInit {

  entities: DTO[];

  constructor(private service: DummyService) { }

  ngOnInit(): void {
    this.service.getAll().subscribe({
        next: (res: DTO[]) => this.entities = res,
        complete: () => console.log(this.entities),
    });
  }

}
```

3.7 Argomenti aggiuntivi

Vediamo ora alcuni argomenti aggiuntivi, che ci permetteranno di capire più a fondo il funzionamento di Angular. Nello specifico, ci concentreremo su:

1. Sincronia

2. Caricamento dei dati
3. File di Configurazione

3.7.1 Sincronia

Come detto nella sezione precedente, l'utilizzo degli Observable offre notevoli vantaggi nella gestione degli eventi e nelle prestazioni, grazie all'utilizzo della programmazione asincrona, consentendo al programma di iniziare task potenzialmente molto lunghe rimanendo comunque responsiva ad altri eventi durante l'esecuzione, senza rallentare l'applicazione in attesa che questa finisca. Questa tecnica di programmazione è in diretto contrasto con la programmazione sincrona, in cui il programma aspetta che una task sia terminata prima di passare a quella successiva.
Nonostante i vantaggi che questo comporti in termini di prestazioni, alle volte vorremmo aspettare che una task termini prima di passare a quella successiva; ad esempio, potremmo voler aspettare il risultato di una chiamata http per evitare di operare con oggetti potenzialmente undefined. Per far ciò, possiamo utilizzare il blocco next del metodo subscribe, nel quale tutte le operazioni vengono eseguite in modo sincono. Vediamo però un metodo più generale per ovviare a questo problema, attuabile ogniqualvolta si abbia a che fare con una funzione asincrona.
Una funzione asincrona è una qualunque funzione che opera in maniera asincrona, utilizzando una Promise implicita per restituire il risultato. Essa viene indicato dalla keyword **async**. All'interno di una tale funzione, possiamo utilizzare l'operatore await per comunicare al programma di attendere l'esito di una Promise prima di passare alla riga di codice successiva. Vediamo un esempio: creiamo artificialmente una funzione che restituisca un apromise al nostro scopo

```
function resolveAfter2Seconds() {
     return new Promise(resolve => {
       setTimeout(() => {
         resolve('Finito');
       }, 2000);
     });
 }


async myMethod() {
     console.log("Effettuo la chiamata");
     var x = await resolveAfter2Seconds();
     console.log(x);
}

myMethod();

//log:
// Effettuo la chiamata
//Finito
```

Se non avessimo utilizzato la keyword async avremmo invece ottenuto come log:
Effettuo la chiamata
[object Promise]
in quanto il programma passa subito alla riga successiva, senza aspettare che la promise venga effettivamente 'mantenuta'.

3.7.2 Caricamento dei dati

In questa sezione tratteremo di come angular carichi i moduli necessari al corretto funzionamento dell'applicazione. Un modulo può essere caricato in due modi:

1. eager: in breve, tali moduli vengono caricati prima che l'application venga lanciata.
2. lazy: al contrario, tali moduli vengono caricati solamente quando c'è il bisogno.

Il modulo principale (AppModule) viene sempre caricato in maniera eager, in quanto responsabile del funzionamento dell'intera app. I restanti moduli vengono invece caricati a discrezione del programmatore. Per specificare il tipo di caricamento di un modulo, abbiamo i seguenti modi:

- Un modulo è caricato in maniera eager di default, di conseguenza si deve semplicemente dichiarare negli import dell'AppModule, come discusso in precedenza.
- Per caricare un modulo in maniera lazy, questo non deve essere dichiarato negli import di AppModule, bensì dobbiamo operare nell'AppRoutingModule (il modulo di routing principale) come segue: quando dichiariamo le routes, aggiungiamo la seguente route, dove al posto della proprietà component specifichiamo la proprietà loadChildren:

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

In questo esempio, './items/items.module' è il path relativo al modulo che vogliamo caricare lazily e ItemsModule è la classe che viene esportata all'interno del file. In seguito, all'interno del RoutingModule del modulo che vogliamo caricare in maniera lazy, aggiungiamo semplicemente le routes per le sue componenti come faremmo normalmente.
Caricare i dati in questo modo è particolarmente utile quando le dimensioni dell'applicazione si fanno relativamente grandi, in modo da non ritardare eccessivamente il lancio dell'applicazione col caricamento preventivo di tutti i moduli.

3.7.3 File di configurazione

Quando creiamo un nuovo progetto da linea di commando, vengono installati automaticamente i package npm di Angular e le dipendenze aggiuntive all'interno di un nuovo workspace. La cartella principale di questo workspace contiene di default vari file di configurazione, all'interno dei quali vengono automaticamente configurate gran parte delle impostazioni utili nello sviluppo. Vediamo ora quali sono questi file e di cosa si occupano:

- **.editorconfig**: contiene le configurazioni relative all'editor di testo che utilizziamo per scrivere il codice.
- **.gitignore**: specifica i file che devono essere ignorati da git.
- **README.md**: contiene una documentazione introduttiva relativa allo scheletro dell'app. Deve essere utilizzato per specificare la documentazione utile all'utilizzo delle varie componenti durante lo sviluppo.
- **angular.json**: contiene le configurazioni di default della cli per tutto il workspace. Al suo interno troviamo ad esempio le configurazioni per le fasi di build e serve, assieme a quelle dei tool di testing utilizzabili dalla cli.
- **package.json**: configura le dipendenze dei pacchetti npm che sono disponibili a tutti i componenti del workspace.
- **package-lock.json**: specifica le versioni di tutti i package installati tramite npm.
- **tsconfig.json**: contiene le configurazioni per il comportamento di TypeScript all'interno del progetto. Ad esempio al suo interno possiamo specificare quanto stretto debba essere il controllo sui tipi oppure come gestire la possibilità che un oggetto sia undefined.