Angular

Angular è un framework open-source per lo sviluppo di applicazioni web, creato e mantenuto da Google. È una delle principali tecnologie utilizzate per la creazione di single-page applications (SPA), ovvero applicazioni web che interagiscono con gli utenti aggiornando dinamicamente la pagina, senza dover ricaricare completamente la pagina stessa.

Ecco alcuni punti chiave su Angular:

Basato su TypeScript:
Angular è scritto interamente in TypeScript, un superset di JavaScript che aggiunge tipizzazione statica opzionale e altre caratteristiche al linguaggio. L'utilizzo di TypeScript fornisce agli sviluppatori un sistema di tipi robusto e aiuta a migliorare la manutenibilità del codice.

Component-based:
Angular è un framework basato su componenti. I componenti sono blocchi autonomi di UI, ciascuno con il proprio comportamento e stile definiti. Questo approccio facilita la creazione di interfacce utente complesse e modulari.

Two-way data binding:
Angular offre un'implementazione di two-way data binding, il che significa che i cambiamenti nei dati del modello possono influenzare direttamente la vista e viceversa. Questo semplifica la gestione dello stato dell'applicazione e l'aggiornamento dell'interfaccia utente in risposta ai cambiamenti dei dati.

Routing:
Angular fornisce un sistema di routing integrato che consente agli sviluppatori di gestire la navigazione all'interno dell'applicazione e di definire percorsi per diverse visualizzazioni.

Iniezione di dipendenza:
Angular utilizza l'iniezione di dipendenza per gestire le dipendenze tra i componenti dell'applicazione. Questo approccio favorisce la modularità e la riusabilità del codice, consentendo agli sviluppatori di scrivere componenti più indipendenti e testabili.

Ampia comunità e supporto:
Grazie al sostegno di Google e alla sua popolarità, Angular ha una vasta comunità di sviluppatori attivi che forniscono supporto, risorse educative, librerie aggiuntive e strumenti per lo sviluppo.

Architettura modulare:

Sono un concetto fondamentale che consente di organizzare e strutturare un'applicazione in modo modulare. Un modulo in Angular è una raccolta di componenti, direttive, servizi e altri artefatti correlati che collaborano per fornire funzionalità specifiche all'interno dell'applicazione.

Ecco alcuni punti chiave sui moduli in Angular:

1. **NgModule**: In Angular, i moduli sono implementati utilizzando la classe `NgModule`, che è un decoratore TypeScript che viene applicato a una classe per definire un modulo. Un modulo viene definito all'interno di un file TypeScript separato e solitamente ha il suffisso `.module.ts`.
2. **Struttura di un modulo**: Un modulo può contenere diversi elementi, inclusi componenti, direttive, servizi, pipe, e altri moduli. La definizione di un modulo specifica quali elementi sono inclusi all'interno di esso e come sono configurati.
3. **Dichiarazioni**: Una delle proprietà fondamentali di un modulo è `declarations`, che elenca tutti i componenti, le direttive e le pipe che appartengono al modulo. Dichiarare un componente all'interno di un modulo lo rende disponibile per l'utilizzo all'interno del template di altri componenti all'interno dello stesso modulo.

4. **Importazioni**: Un modulo può importare altri moduli per utilizzare le funzionalità che forniscono. Questo è realizzato utilizzando la proprietà `imports`. Le importazioni di moduli consentono di organizzare l'applicazione in modo gerarchico e di utilizzare funzionalità definite in altri moduli.
5. **Esportazioni**: Un modulo può esportare determinati elementi, come componenti, direttive o servizi, rendendoli disponibili per essere utilizzati da altri moduli. Le esportazioni sono specificate tramite la proprietà `exports`.
6. **Provider**: La proprietà `providers` di un modulo definisce i servizi che il modulo offre o utilizza. I servizi forniti da un modulo saranno disponibili per l'iniezione di dipendenza all'interno del modulo stesso e di altri moduli che lo importano.
7. **Bootstrap**: Ogni applicazione Angular ha un modulo radice, comunemente chiamato AppModule, che viene avviato durante il bootstrap dell'applicazione. Il modulo radice viene specificato nel file `main.ts` e serve come punto di partenza per l'applicazione Angular.
8. **Lazy loading**: Angular supporta il caricamento lazy dei moduli, consentendo di caricare i moduli solo quando sono richiesti. Ciò può migliorare le prestazioni dell'applicazione riducendo il tempo di caricamento iniziale.

Angular CLI:

Angular fornisce un'interfaccia a riga di comando (CLI) che semplifica la creazione, la gestione e il deployment delle applicazioni Angular. La CLI offre strumenti per generare automaticamente codice, eseguire test, ottimizzare il bundle dell'applicazione e molto altro ancora.

Ecco alcuni dei principali utilizzi e funzionalità di Angular CLI:

1. **Generazione di nuovi progetti**: Angular CLI consente di creare nuovi progetti Angular con facilità, fornendo una struttura di base e i file necessari per iniziare a sviluppare immediatamente.
2. **Generazione di componenti, servizi e altri artefatti**: Con Angular CLI è possibile generare rapidamente nuovi componenti, servizi, direttive, moduli e altri artefatti Angular. Questo velocizza il processo di sviluppo eliminando la necessità di creare manualmente i file e la struttura del codice.

3. **Gestione delle dipendenze**: Angular CLI gestisce automaticamente le dipendenze del progetto, inclusi i pacchetti npm (Node Package Manager). Consente di aggiungere, rimuovere o aggiornare le dipendenze del progetto con facilità utilizzando i comandi appropriati.
4. **Sviluppo locale**: Angular CLI fornisce un server di sviluppo locale che consente di eseguire e testare l'applicazione Angular in ambiente di sviluppo. Il server offre funzionalità come il ricaricamento automatico della pagina (live reload) e la visualizzazione degli errori di compilazione in tempo reale.
5. **Compilazione ottimizzata per la produzione**: Angular CLI supporta la compilazione dell'applicazione in modalità di produzione, ottimizzando il codice per le prestazioni e la dimensione del bundle. Questo include la minimizzazione del codice, la rimozione del codice non utilizzato e altre ottimizzazioni.
6. **Testing automatizzato**: Angular CLI include strumenti per eseguire test automatizzati sull'applicazione Angular, inclusi test unitari e test end-to-end (E2E). Fornisce anche configurazioni predefinite per l'integrazione con framework di test popolari come Jasmine e Karma.
7. **Deployment**: Angular CLI semplifica il processo di deployment dell'applicazione, consentendo di generare un pacchetto ottimizzato per la produzione che può essere distribuito su server web o piattaforme di hosting.

Decoratori

In Angular, i decoratori sono ampiamente utilizzati per annotare classi e definire il comportamento dei vari elementi all'interno di un'applicazione Angular. Ecco alcuni dei decoratori più comuni utilizzati in Angular:

1. `@Component`: Questo decoratore viene utilizzato per definire un componente Angular. Viene utilizzato sopra una classe TypeScript che rappresenta il componente e accetta un oggetto di metadati che definisce le proprietà del componente, come il selettore, il template, lo stile e altro ancora.

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

```
@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent {
  // Proprietà e logica del componente
}
```

2. `@NgModule`: Questo decoratore viene utilizzato per definire un modulo Angular. Viene utilizzato sopra una classe TypeScript che rappresenta il modulo e accetta un oggetto di metadati che definisce le proprietà del modulo, come le dichiarazioni, le importazioni, i providers e altro ancora.

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

@NgModule({
  declarations: [/* Elenco dei componenti dichiarati nel m
odulo */],
  imports: [BrowserModule /* Altri moduli importati */],
  providers: [/* Servizi forniti nel modulo */],
  bootstrap: [/* Componente di avvio dell'applicazione */]
})
export class AppModule { }
```

3. `@Injectable`: Questo decoratore viene utilizzato per definire un servizio Angular. Viene utilizzato sopra una classe TypeScript che rappresenta il servizio e indica che il servizio può essere iniettato come dipendenza in altre classi.

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

@Injectable({
  providedIn: 'root' // Definisce il livello di iniezione
del servizio
```

```
})
export class ExampleService {
  // Logica del servizio
}
```

4. `@Directive`: Questo decoratore viene utilizzato per definire una direttiva Angular. Viene utilizzato sopra una classe TypeScript che rappresenta la direttiva e accetta un oggetto di metadati che definisce il selettore della direttiva e altre proprietà.

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appExampleDirective]'
})
export class ExampleDirective {
  constructor(private elementRef: ElementRef) {
    // Utilizzo di ElementRef per accedere all'elemento DO
M
  }
}
```

Direttive

Sono uno degli elementi fondamentali di Angular che consentono di estendere e modificare il comportamento del DOM (Document Object Model) all'interno di un'applicazione Angular. Esse forniscono un modo per aggiungere comportamenti specifici o manipolare la struttura del DOM in base alla logica definita nelle classi TypeScript.

Esistono due tipi principali di direttive in Angular:

1. **Direttive strutturali**: Le direttive strutturali modificano la struttura del DOM aggiungendo, rimuovendo o sostituendo elementi HTML basati su condizioni logiche. Le direttive strutturali più comuni sono `ngIf` e `ngFor`.

- `ngIf`: Consente di condizionare la visualizzazione di un elemento HTML in base a una espressione booleana.

```
<div *ngIf="mostraElemento">
  Contenuto visualizzato se mostraElemento è true.
</div>
```

- `ngFor`: Consente di iterare su una collezione di elementi e generare dinamicamente elementi HTML per ciascun elemento nella collezione.

```
<ul>
  <li *ngFor="let elemento of elementi">
    {{ elemento }}
  </li>
</ul>
```

2. **Direttive attributo**: Le direttive attributo aggiungono o modificano il comportamento di un elemento HTML aggiungendo attributi personalizzati. Le direttive attributo possono essere utilizzate per creare comportamenti riutilizzabili che possono essere applicati a più elementi.

   Ad esempio, si può definire una direttiva attributo personalizzata `appHighlight` che evidenzia il testo di un elemento quando ci si passa sopra con il mouse:

```
import { Directive, ElementRef, HostListener } from '@angu
lar/core';

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

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

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

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

Questa direttiva può essere utilizzata su un elemento HTML come segue:

```
<p appHighlight>
  Passa il mouse su di me per evidenziare.
</p>
```

Le direttive sono un meccanismo potente che consente agli sviluppatori di aggiungere comportamenti personalizzati e manipolare il DOM in modo dichiarativo all'interno delle applicazioni Angular. Possono essere utilizzate per migliorare la modularità, la riusabilità e la manutenibilità del codice.

Componenti

I componenti sono blocchi fondamentali per la costruzione di applicazioni in Angular. Un componente in Angular è una classe TypeScript che interagisce con un file di template HTML e altri file di stile e metadati, per creare un elemento UI riutilizzabile e autosufficiente.

Ecco una panoramica dei componenti in Angular:

1. **Struttura di un componente**:
   - **Classe TypeScript**: Contiene la logica del componente, come proprietà e metodi.
   - **Template HTML**: Definisce la struttura e il layout del componente, includendo HTML e direttive Angular.

- **File di stile**: Opzionale, definisce lo stile CSS per il componente.
- **Metadati del componente**: Vengono definiti utilizzando il decoratore `@Component` e includono informazioni come il selettore del componente, il percorso del file di template e del file di stile, e altri metadati.

2. **Decoratore** `@Component`:
   - Il decoratore `@Component` viene utilizzato per annotare la classe del componente e fornisce metadati che Angular utilizza per configurare e gestire il componente.
   - Questi metadati includono il selettore (per identificare il componente all'interno del template HTML), il percorso del file del template, i percorsi dei file di stile, i providers di servizi specifici per il componente e altre opzioni.
3. **Ciclo di vita del componente**:
   - I componenti in Angular seguono un ciclo di vita predefinito, composto da eventi che si verificano durante la creazione, l'aggiornamento e la distruzione del componente.
   - Alcuni dei metodi più comuni del ciclo di vita includono `ngOnInit`, `ngOnChanges`, `ngAfterViewInit`, `ngOnDestroy`, che vengono chiamati in momenti specifici durante il ciclo di vita del componente.
4. **Comunicazione tra componenti**:
   - I componenti possono comunicare tra loro attraverso l'utilizzo di input e output.
   - Gli input consentono di passare dati da un componente padre a un componente figlio, mentre gli output consentono al componente figlio di comunicare con il componente padre attraverso eventi.
5. **Iniezione di dipendenza**:
   - I componenti possono utilizzare l'iniezione di dipendenza per accedere ai servizi o ad altre dipendenze necessarie per la loro logica.
   - Questo approccio favorisce la separazione delle responsabilità e la modularità del codice.

Supponiamo di avere un componente `HelloComponent` che accetta un nome come input e visualizza un saluto personalizzato.

```
// hello.component.ts
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent {
  @Input() name: string;

  constructor() { }
}
```

```
<!-- hello.component.html -->
<div>
  <h2>Hello, {{ name }}!</h2>
</div>
```

```
/* hello.component.css */
h2 {
  color: blue;
}
```

Template

In Angular è un file HTML che definisce la struttura e il layout dell'interfaccia utente di un componente. I template vengono utilizzati per definire come i dati vengono presentati agli utenti e per interagire con essi.

In pratica, un template contiene HTML, direttive Angular e sintassi specifiche di Angular (come `*ngFor`, `*ngIf`, interpolazione `{{ }}`, ecc.) per creare componenti dinamici e reattivi.

Ecco alcuni concetti chiave relativi ai template in Angular:

1. **Interpolazione**: La sintassi di interpolazione `{{ }}` consente di incorporare valori delle proprietà del componente direttamente nel template HTML. Ad esempio, `{{ user.name }}` verrà sostituito con il valore della proprietà `name` dell'oggetto `user`.
2. **Direttive**: Le direttive Angular consentono di aggiungere comportamenti dinamici agli elementi HTML. Ad esempio, `ngFor` e `ngIf` sono direttive strutturali che consentono di iterare su una lista e di condizionare la visualizzazione di elementi HTML in base a condizioni booleane.
3. **Event Binding**: È possibile collegare eventi HTML come `click`, `input`, `change`, ecc., a metodi definiti nel componente utilizzando la sintassi di event binding. Ad esempio, `(click)="onButtonClick()"` chiamerà il metodo `onButtonClick()` del componente quando viene fatto clic su un elemento HTML.
4. **Two-Way Data Binding**: Angular supporta il two-way data binding, che permette di sincronizzare automaticamente i dati tra il modello e la vista. Questo significa che le modifiche fatte nel template (ad esempio, tramite un input utente) possono essere riflesse automaticamente nel modello del componente e viceversa.
5. **Direttive personalizzate**: È possibile creare direttive personalizzate per estendere le funzionalità degli elementi HTML esistenti o per creare nuovi comportamenti. Le direttive personalizzate vengono utilizzate nel template esattamente come le direttive predefinite di Angular.

Pipes

Le pipes in Angular sono degli strumenti molto utili per la trasformazione dei dati all'interno dei template HTML. Le pipes consentono di trasformare i valori prima di visualizzarli nel template o di eseguire operazioni di formattazione sui dati. Le pipes vengono utilizzate direttamente all'interno dei template HTML, senza richiedere codice TypeScript aggiuntivo.

Ecco alcuni esempi comuni di pipes in Angular:

1. `{{ expression | uppercase }}`:
   - Questa pipe trasforma una stringa in maiuscolo.
2. `{{ expression | lowercase }}`:
   - Questa pipe trasforma una stringa in minuscolo.
3. `{{ expression | currency }}`:
   - Questa pipe formatta un numero come una valuta utilizzando la valuta corrente.
4. `{{ expression | date }}`:
   - Questa pipe formatta una data in base al formato specificato.
5. `{{ expression | percent }}`:
   - Questa pipe converte un numero in una stringa percentuale.
6. `{{ expression | number }}`:
   - Questa pipe formatta un numero in base al formato specificato.
7. `{{ expression | slice:start:end }}`:
   - Questa pipe restituisce una sottostringa di una stringa, iniziando dall'indice `start` fino all'indice `end` (escluso).
8. `{{ expression | async }}`:
   - Questa pipe gestisce automaticamente i valori asincroni, consentendo di visualizzare i risultati una volta disponibili.

È anche possibile concatenare più pipes per eseguire più trasformazioni sui dati, ad esempio:

```
{{ user.name | uppercase | slice:0:5 }}
```

Questa espressione converte il nome dell'utente in maiuscolo e quindi restituisce solo i primi cinque caratteri.

Inoltre, è possibile creare pipes personalizzate per adattare le proprie esigenze specifiche. Le pipes personalizzate sono classi TypeScript annotate con `@Pipe` e implementano l'interfaccia `PipeTransform`. Questo permette di definire la logica di trasformazione personalizzata e renderla disponibile all'interno dei template Angular.

In generale, le pipes offrono un modo semplice e potente per manipolare e formattare i dati all'interno dei template Angular, rendendo più facile la presentazione dei dati agli utenti in modo leggibile e comprensibile.

Lifecycle dei componenti

Rappresenta una serie di fasi attraverso le quali passa un componente durante la sua esistenza, dalla creazione alla distruzione. Queste fasi consentono ai programmatori di eseguire operazioni specifiche in momenti specifici durante il ciclo di vita del componente, come l'inizializzazione dei dati, l'aggiornamento dell'interfaccia utente o la gestione delle risorse.

Ecco le principali fasi del ciclo di vita di un componente in Angular:

1. **ngOnChanges**: Questo metodo viene chiamato ogni volta che cambia un'input del componente. È utile per reagire ai cambiamenti nei dati di input e prendere azioni di conseguenza.
2. **ngOnInit**: Questo metodo viene chiamato una volta dopo che il componente è stato inizializzato e tutte le sue direttive figlio sono state inizializzate. È comunemente utilizzato per inizializzare i dati del componente o per eseguire operazioni una tantum.
3. **ngAfterContentInit**: Questo metodo viene chiamato dopo che il contenuto proiettato (tramite le direttive `ng-content`) è stato inizializzato. È utile per eseguire operazioni che richiedono l'accesso al contenuto proiettato del componente.
4. **ngAfterViewInit**: Questo metodo viene chiamato dopo che la vista del componente e le sue direttive figlio sono state inizializzate. È comunemente utilizzato per eseguire operazioni che richiedono l'accesso al DOM del componente.

5. **ngOnDestroy**: Questo metodo viene chiamato prima che il componente venga distrutto e rimosso dal DOM. È utile per eseguire operazioni di pulizia, come l'annullamento di sottoscrizioni a osservabili o la disconnessione da servizi.

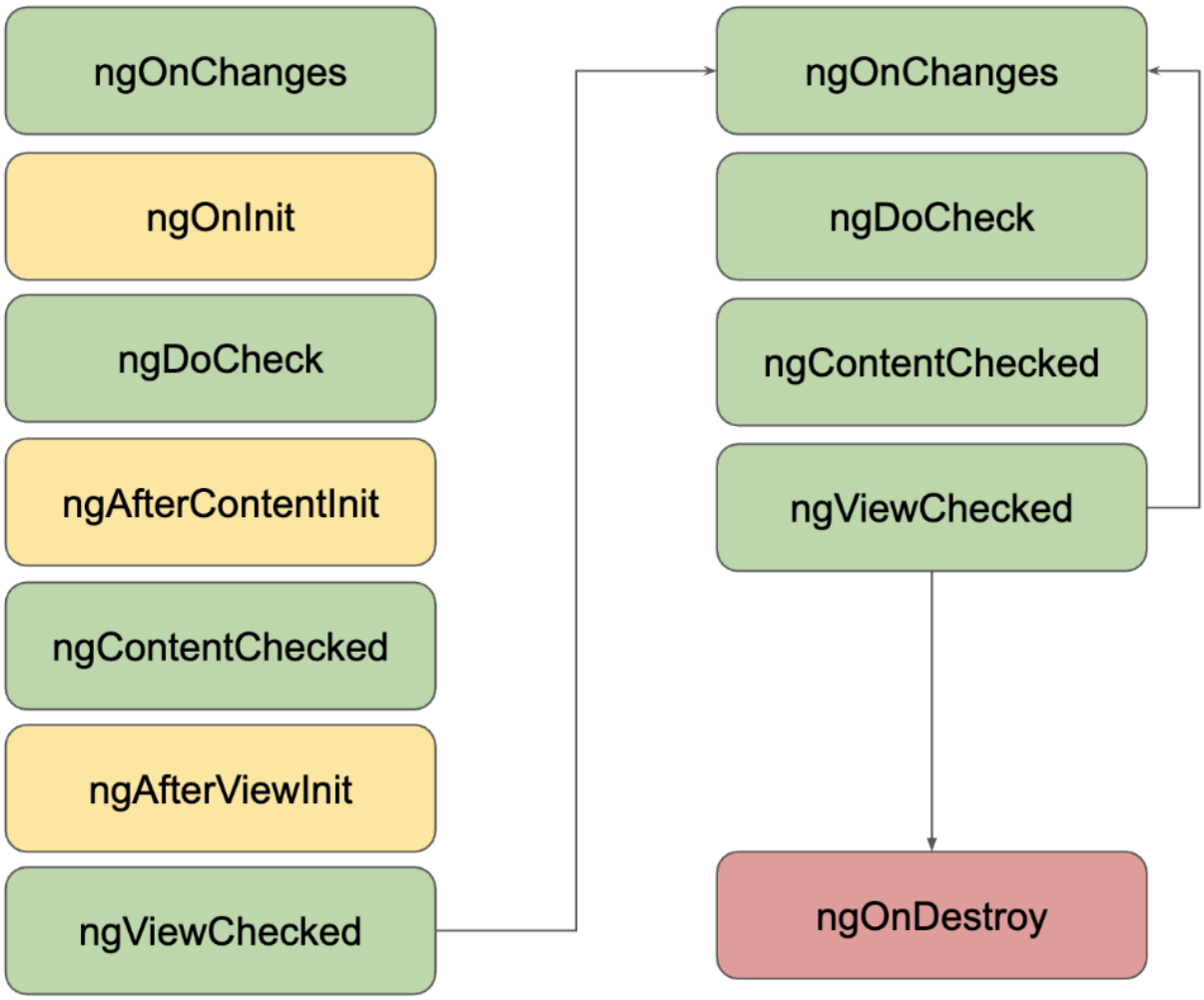
Durante il ciclo di vita del componente, è possibile sfruttare queste fasi per eseguire azioni specifiche in momenti opportuni. Ad esempio, l'inizializzazione dei dati del componente può essere eseguita in `ngOnInit`, mentre la pulizia delle risorse può essere eseguita in `ngOnDestroy`.

Ecco un esempio di come potrebbe apparire l'implementazione di queste fasi nel codice TypeScript di un componente Angular:

```
import { Component, OnInit, OnDestroy, OnChanges, AfterConten
tInit, AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent implements OnInit, OnDestroy, O
nChanges, AfterContentInit, AfterViewInit {

  constructor() { }

  ngOnInit(): void {
    // Inizializzazione dei dati del componente
  }

  ngOnChanges(): void {
    // Risponde ai cambiamenti nei dati di input
  }

  ngAfterContentInit(): void {
    // Eseguito dopo l'inizializzazione del contenuto proiett
ato
  }
```

```
  ngAfterViewInit(): void {
    // Eseguito dopo l'inizializzazione della vista del compo
nente
  }

  ngOnDestroy(): void {
    // Pulizia delle risorse del componente prima della distr
uzione
  }
}
```

Implementare i metodi corrispondenti per ciascuna fase del ciclo di vita del componente consente di controllare efficacemente il comportamento e le operazioni del componente in base alle esigenze specifiche dell'applicazione.



Si può passare dati da un componente padre a un componente figlio utilizzando gli input. Gli input consentono al componente padre di passare dati al componente figlio come proprietà.

Ecco come è possibile passare dati ai componenti figli utilizzando gli input:

1. **Definizione dell'input nel componente figlio**:
   Nel componente figlio, definire una o più proprietà decorate con l'annotazione `@Input()` per indicare che possono ricevere dati dal componente padre.

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

@Component({
```

```
  selector: 'app-child',
  template: '<p>Child Component - Received Data: {{ receiv
edData }}</p>'
})
export class ChildComponent {
  @Input() receivedData: string;
}
```

2. **Passaggio dei dati dal componente padre al componente figlio**:
   Nel template del componente padre, utilizzare la direttiva `[property]` per associare una proprietà del componente figlio a una proprietà del componente padre. Questo permette di passare i dati al componente figlio.

   ```
   <app-child [receivedData]="parentData"></app-child>
   ```

   Nel controller del componente padre, assegnare un valore alla proprietà che si desidera passare al componente figlio.

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

   @Component({
     selector: 'app-parent',
     template: '<app-child [receivedData]="parentData"></app-
   child>'
   })
   export class ParentComponent {
     parentData = 'Hello from Parent';
   }
   ```

In questo modo, il valore della proprietà `parentData` del componente padre viene passato al componente figlio come `receivedData`. Ogni volta che il valore di `parentData` nel componente padre cambia, il valore di `receivedData` nel componente figlio viene aggiornato automaticamente.

È possibile passare qualsiasi tipo di dato, non solo stringhe, utilizzando input nei componenti Angular. Questo approccio consente una comunicazione

unidirezionale dei dati dai componenti padre ai componenti figlio, che è una delle pratiche consigliate in Angular per mantenere le applicazioni più coerenti e prevedibili.

Per ricevere invece eventi dai componenti figli, è possibile utilizzare l'output. L'output consente ai componenti figli di emettere eventi che possono essere catturati e gestiti dal componente padre.

Ecco come è possibile ricevere eventi dai componenti figli utilizzando l'output:

1. **Definizione dell'output nel componente figlio**:
   Nel componente figlio, definire un evento personalizzato utilizzando l'annotazione `@Output()` e `EventEmitter`. L'evento personalizzato può essere emesso quando si verifica un'azione specifica nel componente figlio.

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

@Component({
  selector: 'app-child',
  template: '<button (click)="emitEvent()">Click Me</butto
n>'
})
export class ChildComponent {
  @Output() customEvent = new EventEmitter<void>();

  emitEvent() {
    this.customEvent.emit();
  }
}
```

2. **Cattura dell'evento nel componente padre**:
   Nel template del componente padre, utilizzare la direttiva `(evento)` per catturare l'evento emesso dal componente figlio e associarlo a un metodo del componente padre.

```
<app-child (customEvent)="handleCustomEvent()"></app-child
>
```

Nel controller del componente padre, definire il metodo `handleCustomEvent()` per gestire l'evento ricevuto dal componente figlio.

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

@Component({
  selector: 'app-parent',
  template: '<app-child (customEvent)="handleCustomEvent
()"></app-child>'
})
export class ParentComponent {
  handleCustomEvent() {
    console.log('Custom Event Received from Child');
  }
}
```

In questo modo, quando l'evento personalizzato viene emesso dal componente figlio, il metodo `handleCustomEvent()` nel componente padre viene chiamato e può gestire l'evento come desiderato.

Questo approccio consente una comunicazione unidirezionale dei dati e degli eventi dai componenti figli ai componenti genitori, che è un'approccio comune e una pratica consigliata in Angular per mantenere le applicazioni organizzate e prevedibili.

Two-way-data-binding

Il two-way data-binding è un'operazione bidirezionale che permette di mantenere sincronizzati i dati tra il modello (componente TypeScript) e la vista (template HTML). Questo significa che le modifiche effettuate nel modello si riflettono automaticamente nella vista e viceversa, senza dover scrivere codice aggiuntivo per aggiornare manualmente i dati.

Può essere realizzato utilizzando la direttiva `ngModel`, che combina il binding delle proprietà con il binding degli eventi, consentendo di sincronizzare i dati tra un elemento HTML di input e una proprietà del modello del componente.

Ecco come implementare il two-way data-binding con `ngModel`:

1. **Importare il modulo `FormsModule`**:
   Per utilizzare `ngModel`, è necessario importare il modulo `FormsModule` nell'`AppModule` o in qualsiasi altro modulo che utilizzi il two-way data-binding.

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule],
  // altri metadati del modulo
})
export class AppModule { }
```

2. **Utilizzare `ngModel` nell'elemento HTML**:
   Aggiungere l'attributo `[(ngModel)]` a un elemento HTML di input, come un `<input>` o un `<textarea>`, e associarlo a una proprietà del modello del componente tramite una direttiva di binding.

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

3. **Gestire i dati nel modello del componente**:
   Nel controller del componente (il file TypeScript), definire una proprietà per memorizzare i dati e inizializzarla con un valore di default.

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

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

```
})
export class ExampleComponent {
  nomeUtente: string = '';
}
```

Interazione padre-figli tramite variabili locali

Le interazioni tra componenti padre e figli possono anche avvenire tramite variabili locali e referenze di elementi HTML. Le variabili locali consentono al componente padre di accedere direttamente a un componente figlio o a un elemento HTML all'interno del template del componente padre.

Ecco come è possibile utilizzare variabili locali per le interazioni tra componente padre e figlio:

1. **Assegnazione di una variabile locale a un componente figlio**:
   Utilizzare il simbolo
   `#` all'interno del template del componente padre per creare una variabile locale che fa riferimento a un componente figlio o a un elemento HTML all'interno del template del componente padre.

   ```
   <!-- Nel template del componente padre -->
   <app-child #childComponent></app-child>
   ```

2. **Accesso alla variabile locale nel controller del componente padre**:
   Utilizzare
   `@ViewChild` o `@ViewChildren` per accedere alla variabile locale all'interno del controller del componente padre.

   ```
   import { Component, ViewChild } from '@angular/core';
   import { ChildComponent } from './child.component'; // Imp
   ortare il componente figlio

   @Component({
     selector: 'app-parent',
     templateUrl: './parent.component.html',
   ```

```
  styleUrls: ['./parent.component.css']
})
export class ParentComponent {
  @ViewChild('childComponent') childComponent: ChildCompon
ent; // Accesso alla variabile locale del componente figli
o

  // Metodo nel controller del componente padre per chiama
re un metodo nel componente figlio
  callChildMethod() {
    this.childComponent.methodInChildComponent();
  }
}
```

3. **Accesso alla variabile locale tramite elementi HTML**:
   È anche possibile utilizzare variabili locali per fare riferimento direttamente agli elementi HTML all'interno del template del componente padre.

   ```
   <!-- Nel template del componente padre -->
   <input type="text" #inputElement>
   ```

   Successivamente, è possibile accedere all'elemento HTML nel controller del componente padre utilizzando `@ViewChild`.

Attribute Directive

Le direttive di attributo (Attribute Directives) in Angular sono un tipo di direttive che vengono utilizzate per modificare l'aspetto o il comportamento degli elementi DOM a cui vengono applicate. Queste direttive vengono applicate a un elemento tramite un attributo.

Ecco alcuni punti chiave sulle direttive di attributo:

1. **Utilizzo**:
   Le direttive di attributo vengono applicate a un elemento HTML aggiungendo

un attributo speciale all'elemento stesso. Quando Angular trova un elemento con un attributo associato a una direttiva di attributo, applica la logica della direttiva a quell'elemento.

2. **Modifica del comportamento**:
   Le direttive di attributo possono modificare il comportamento degli elementi DOM a cui vengono applicate. Ad esempio, la direttiva `ngClass` può essere utilizzata per aggiungere o rimuovere classi CSS in base a determinate condizioni.
3. **Modifica dell'aspetto**:
   Le direttive di attributo possono anche modificare l'aspetto degli elementi DOM. Ad esempio, la direttiva `ngStyle` consente di applicare stili CSS dinamicamente agli elementi in base a determinate condizioni.
4. **Creazione di direttive personalizzate**:
   È possibile creare direttive di attributo personalizzate per estendere le funzionalità di Angular o per aggiungere comportamenti personalizzati agli elementi DOM.
5. **Componenti vs. direttive di attributo**:
   A differenza dei componenti, che vengono utilizzati per creare elementi UI completi, le direttive di attributo vengono utilizzate per modificare l'aspetto o il comportamento degli elementi esistenti senza creare nuovi elementi.

Ecco un esempio di come si utilizza una direttiva di attributo incorporata come `ngStyle`:

```
<div [ngStyle]="{ 'color': 'red', 'font-size': '20px' }">Hell
o</div>
```

Structural Directive

Sono un tipo speciale di direttive utilizzate per modificare la struttura del DOM aggiungendo, rimuovendo o sostituendo elementi DOM in base a determinate condizioni o iterazioni su insiemi di dati.

Ecco una panoramica delle principali direttive strutturali in Angular:

1. **ngIf**:
   La direttiva
   `ngIf` viene utilizzata per condizionare la presenza di un elemento nel DOM. Se l'espressione associata a `ngIf` è valutata come `true`, l'elemento viene inserito nel DOM; altrimenti, viene rimosso.

```
<div *ngIf="isVisible">Elemento visibile</div>
```

2. **ngFor**:
   La direttiva
   `ngFor` viene utilizzata per iterare su una collezione di dati e generare elementi HTML ripetuti in base a ciascun elemento nella collezione.

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

3. **ngSwitch**:
   La direttiva
   `ngSwitch` viene utilizzata per eseguire un'operazione di commutazione su un'espressione e in base al risultato di questa espressione, la direttiva `ngSwitchCase` corrispondente viene resa visibile.

```
<div [ngSwitch]="color">
  <div *ngSwitchCase="'red'">Il colore è rosso</div>
  <div *ngSwitchCase="'green'">Il colore è verde</div>
  <div *ngSwitchCase="'blue'">Il colore è blu</div>
  <div *ngSwitchDefault>Il colore non è rosso, verde o blu
</div>
</div>
```

Dependency Injection e Service

La Dependency Injection (**DI**) è un concetto fondamentale che consente di fornire le dipendenze necessarie a un componente, un servizio o un'altra classe durante la creazione di un'istanza. Questo approccio favorisce la modularità, la manutenibilità e la testabilità del codice, in quanto le dipendenze esterne possono essere facilmente sostituite o mockate durante i test.

Un servizio in Angular è una classe TypeScript annotata con il decoratore `@Injectable()` che può essere iniettata in altri componenti, direttive, servizi o moduli. I servizi sono utilizzati per la logica di business, la comunicazione con il backend, la gestione dello stato dell'applicazione e altre operazioni non legate alla visualizzazione.

Ecco come funziona la Dependency Injection e come i servizi vengono utilizzati in Angular:

1. **Definizione del servizio**:
   Creare una classe TypeScript per il servizio e annotarla con `@Injectable()` per indicare che il servizio è iniettabile.

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

   @Injectable({
     providedIn: 'root', // Opzionale: specifica il livello d
   i iniezione del servizio (radice dell'app)
   })
   export class DataService {
     getData(): string {
       return 'Dati ottenuti dal servizio';
     }
   }
   ```

2. **Iniezione del servizio**:
   Iniettare il servizio nei componenti, direttive o altri servizi che ne hanno bisogno nel loro costruttore.

```
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-example',
  template: '<p>{{ data }}</p>',
})
export class ExampleComponent {
  data: string;

  constructor(private dataService: DataService) {}

  ngOnInit(): void {
    this.data = this.dataService.getData();
  }
}
```

3. **Utilizzo del servizio**:
   Ora il servizio può essere utilizzato all'interno del componente (o di altre classi) per accedere ai suoi metodi e dati.

```
<!-- Nel template del componente -->
<p>{{ data }}</p>
```

La Dependency Injection in Angular gestisce automaticamente la creazione e l'iniezione delle istanze dei servizi. È possibile iniettare un servizio su più livelli di applicazione (ad esempio, a livello di componente, di modulo o di applicazione) specificando il livello di iniezione appropriato nel decoratore `@Injectable()` o nei provider del modulo.

Utilizzare i servizi in Angular consente di mantenere il codice separato e di evitare la duplicazione di logica all'interno dei componenti. Inoltre, i servizi sono un ottimo strumento per la condivisione di dati e la gestione dello stato tra componenti diversi.

Definire dipendenze per DI

Le dipendenze per la Dependency Injection (DI) vengono definite in vari modi a seconda delle esigenze specifiche dell'applicazione e delle componenti coinvolte. Ecco alcuni metodi comuni per definire le dipendenze:

1. **Utilizzo del costruttore**:
   Il metodo più comune per definire le dipendenze è attraverso il costruttore delle classi. Le dipendenze vengono dichiarate come parametri del costruttore e Angular si occupa di fornire automaticamente le istanze dei servizi richiesti al momento dell'iniezione.

   ```
   import { Injectable } from '@angular/core';
   import { HttpClient } from '@angular/common/http';

   @Injectable({
     providedIn: 'root',
   })
   export class DataService {
     constructor(private http: HttpClient) {}

     // Metodi che utilizzano HttpClient per interagire con u
   n backend
   }
   ```

2. **Utilizzo di `@Inject`**:
   In alcuni casi, potresti avere la necessità di fornire una dipendenza specifica manualmente. Puoi farlo utilizzando il decoratore `@Inject` insieme al costruttore.

   ```
   import { Injectable, Inject } from '@angular/core';
   import { MY_CUSTOM_TOKEN } from './my-custom-token';

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

```
  constructor(@Inject(MY_CUSTOM_TOKEN) private myCustomDep
endency: any) {}

  // Altri metodi del servizio
}
```

3. **Provider del modulo**:
   È possibile definire le dipendenze anche all'interno dei provider del modulo. Questo è utile quando si desidera fornire lo stesso servizio in più parti dell'applicazione.

```
import { NgModule } from '@angular/core';
import { MyService } from './my.service';

@NgModule({
  providers: [MyService],
})
export class MyModule {}
```

4. **Provider personalizzati**:
   È possibile creare provider personalizzati per gestire la creazione e la fornitura delle istanze di servizio. Questo è utile per casi particolari in cui è necessario personalizzare il comportamento della DI.

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

@Injectable({
  providedIn: 'root',
  useFactory: () => {
    // Logica personalizzata per la creazione dell'istanza
del servizio
    return new MyCustomService();
  },
})
export class MyService {}
```

Comunicazione fra componenti tramite Service

La comunicazione tra componenti tramite un servizio è un modo efficace per condividere dati e funzionalità tra componenti diversi in Angular. Utilizzando un servizio come intermediario, è possibile evitare la dipendenza diretta tra i componenti, mantenendo così un'accoppiamento più basso e una maggiore modularità nell'applicazione.

Ecco come è possibile implementare la comunicazione tra componenti tramite un servizio in Angular:

1. **Creazione del servizio**:
   Inizia creando un servizio che sarà responsabile per la comunicazione tra i componenti. Questo servizio dovrebbe contenere metodi per inviare e ricevere dati o eventi.

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class DataService {
  private dataSubject = new Subject<any>();
  data$ = this.dataSubject.asObservable();

  sendData(data: any) {
    this.dataSubject.next(data);
  }
}
```

2. **Utilizzo del servizio nei componenti**:
   Inietta il servizio nei componenti che devono comunicare tra loro. Ogni componente può utilizzare i metodi del servizio per inviare o ricevere dati.

```
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-sender',
  template: '<button (click)="sendData()">Send Data</butto
n>',
})
export class SenderComponent {
  constructor(private dataService: DataService) {}

  sendData() {
    this.dataService.sendData('Hello from Sender');
  }
}
```

```
import { Component } from '@angular/core';
import { DataService } from './data.service';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-receiver',
  template: '<p>{{ receivedData }}</p>',
})
export class ReceiverComponent {
  receivedData: any;
  private subscription: Subscription;

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.subscription = this.dataService.data$.subscribe(d
ata => {
      this.receivedData = data;
```

```
    });
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}
```

Forms

La gestione dei form è un'operazione fondamentale per interagire con gli utenti e raccogliere dati all'interno delle applicazioni. Angular offre diverse funzionalità e strumenti per gestire i form in modo efficace e robusto.

Ecco una panoramica delle funzionalità dei form in Angular:

1. **Template-driven forms**:
   I template-driven forms sono il modo più semplice per gestire i form in Angular. Si basano principalmente sull'utilizzo di direttive nel template HTML per definire e validare i campi del form.
2. **Reactive forms**:
   I reactive forms sono una modalità più avanzata per gestire i form in Angular. Si basano sull'utilizzo di oggetti JavaScript per definire il modello dei form e offrono un maggiore controllo e flessibilità rispetto ai template-driven forms.
3. **Validazione dei form**:
   Angular offre un sistema di validazione dei form potente e flessibile, che consente di validare i campi del form sia lato client che lato server.
4. **Submit dei form**:
   Angular fornisce meccanismi per gestire l'evento di invio dei form, sia attraverso il binding agli eventi nativi HTML (come `(submit)`) che attraverso metodi nel controller del componente.
5. **Gestione degli errori dei form**:
   È possibile gestire e visualizzare errori nei form sia durante la

compilazione che durante l'invio. Angular fornisce strumenti per visualizzare messaggi di errore personalizzati e per controllare lo stato di validità complessivo del form.

6. **Comunicazione con il backend**:
   Angular facilita la comunicazione con il backend per inviare e ricevere dati dai form. È possibile utilizzare servizi HTTP per inviare i dati del form al server e ricevere eventuali risposte.

Ecco un esempio di un semplice form template-driven in Angular:

```
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <div>
    <label for="name">Name</label>
    <input type="text" id="name" name="name" [(ngModel)]
="formData.name" required>
  </div>
  <div>
    <label for="email">Email</label>
    <input type="email" id="email" name="email" [(ngMode
l)]="formData.email" required email>
  </div>
  <button type="submit">Submit</button>
</form>
```

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

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent {
  formData = {
    name: '',
    email: ''
```

```
  };

  onSubmit(form: NgForm) {
    if (form.valid) {
      console.log('Form submitted:', this.formData);
    } else {
      console.error('Form is invalid');
    }
  }
}
```

RxJs

RxJS, acronimo di Reactive Extensions for JavaScript, è una libreria per la programmazione reattiva in JavaScript che offre un'implementazione della specifica Observables, che consente di gestire e manipolare sequenze di eventi asincroni. RxJS è ampiamente utilizzato in Angular per gestire eventi, effettuare chiamate HTTP, elaborare dati e molto altro ancora.

Ecco alcuni concetti chiave di RxJS:

1. **Observable**:
   Un Observable rappresenta una sequenza di eventi che possono essere osservati nel tempo. Un Observable può emettere valori, errori e completamenti.
2. **Observer**:
   Un Observer è un oggetto che osserva un Observable e risponde agli eventi emessi da esso. Gli Observer possono ricevere valori emessi, gestire errori e reagire al completamento dell'Observable.
3. **Operatori**:
   Gli operatori in RxJS sono funzioni che consentono di manipolare e trasformare i dati emessi da un Observable. Gli operatori possono essere utilizzati per filtrare, mappare, ridurre, combinare e molto altro ancora.

4. **Subscription**:
   Una Subscription rappresenta l'esecuzione di un Observable. È possibile sottoscriversi a un Observable per iniziare l'osservazione degli eventi e utilizzare la Subscription per annullare l'osservazione quando non è più necessaria.

5. **Subject**:
   Un Subject è sia un Observer che un Observable. Può essere utilizzato per inviare valori multipli a più Observer e rappresenta una sorta di "canale" per la comunicazione tra parti di un'applicazione.

6. **Pipeable Operators**:
   I Pipeable Operators sono una forma di operatori che possono essere concatenati insieme utilizzando il metodo `pipe()`. Questo permette di creare catene di operazioni più leggibili e componibili.

Ecco un esempio semplice di utilizzo di RxJS per creare un Observable che emette una sequenza di numeri:

```
import { Observable, interval } from 'rxjs';

const observable = new Observable<number>(observer => {
  let count = 0;
  const intervalId = setInterval(() => {
    observer.next(count++);
  }, 1000);

  return () => clearInterval(intervalId);
});

const subscription = observable.subscribe(value => consol
e.log(value));

setTimeout(() => {
  subscription.unsubscribe();
}, 5000);
```