Introduzione ad Angular

Riferimento	Angular
Versione	2.0.0
Data	12/12/24
Destinatario	TMs
Presentato da	Raffaella Spagnuolo, Alessia Ture
Approvato da	Raffaella Spagnuolo, Alessia Ture

Revision History

Data	Versione	Descrizione	Autori
14/11/24	1.0.0	Prima Stesura	ATu
12/12/24	2.0.0	Revisione	RS

Project Managers

Nome	Acronimo	Contatto
Raffaella Spagnuolo	RS	r.spagnuolo6@studenti.unisa.it
Alessia Ture	ATu	a.ture@studenti.unisa.it

Sommario

1 Introduzione

1.1 Scopo del documento

Questo documento è stato creato per fornire una guida completa e dettagliata sui concetti avanzati di Angular, includendo anche una revisione dei concetti di base. È destinato ai membri del team come supporto nello sviluppo dell'applicazione, facilitando la comprensione delle funzionalità fondamentali e avanzate di Angular per un uso efficace e strutturato.

1.2 Obiettivi

L'obiettivo principale di questo documento è aiutare il team a padroneggiare e applicare i concetti di Angular in modo efficiente. Include una panoramica approfondita su temi chiave come il routing, la gestione delle entità, l'utilizzo dei servizi, gli Observable e le pratiche essenziali come la gestione dei componenti, dei moduli e il data binding.

2 Concetti Base di Angular

2.1 Componenti

Angular è basato su componenti e moduli. I **componenti** sono unità di interfaccia utente che rappresentano sezioni specifiche dell'applicazione. In Angular, i componenti sono elementi fondamentali dell'architettura del framework, utilizzati per creare e organizzare l'interfaccia utente di un'applicazione. Ogni componente rappresenta una parte specifica dell'interfaccia e svolge un ruolo ben definito all'interno dell'applicazione. Ad esempio, in un e-commerce, un componente potrebbe rappresentare una scheda prodotto, mentre un altro potrebbe gestire la visualizzazione del profilo utente.

Un componente in Angular è costituito da tre elementi principali:

- **Template HTML:** Definisce la struttura e il contenuto visibile del componente. Il template contiene il codice HTML che verrà visualizzato e può includere anche binding di dati e direttive per manipolare dinamicamente la visualizzazione;
- **Classe TypeScript:** Definisce la logica del componente e contiene i dati e le funzioni che supportano il template. La classe contiene:
    - **Proprietà:** Variabili o dati utilizzati nel template. Queste proprietà definiscono lo stato del componente e sono utilizzate per gestire le informazioni visualizzate o elaborate;
    - **Metodi:** Funzioni che svolgono azioni o calcoli specifici. I metodi sono spesso invocati da eventi nel template, come clic o modifiche di input, e sono utilizzati per aggiornare il contenuto del componente o reagire a interazioni dell'utente;
- **Foglio di Stile CSS:** Definisce lo stile del componente. Ogni componente può avere stili CSS specifici per personalizzare il design e il layout. In Angular, ogni componente può essere isolato nel suo stile grazie al meccanismo di "encapsulation", che evita che i suoi stili influenzino altri componenti.

2.1.1 Ciclo di Vita di un Componente

Un componente Angular passa attraverso diverse fasi durante la sua vita, note come hook del ciclo di vita. Questi hook permettono di eseguire operazioni specifiche in vari momenti del ciclo di vita del componente:

- **ngOnInit():** Chiamato una volta quando il componente viene inizializzato. È utile per eseguire il codice di inizializzazione, come il recupero di dati;

- **ngOnChanges():** Chiamato ogni volta che una proprietà di input cambia, permettendo di rispondere ai cambiamenti nei dati passati al componente;
- **ngOnDestroy():** Chiamato una volta prima che il componente venga distrutto, permettendo di rilasciare risorse e annullare le subscription.

2.1.2 Creazione di un Componente

Per creare un nuovo componente in IntelliJ IDEA, segui questi passaggi:

1. Fai clic destro sulla cartella in cui desideri creare il componente (ad esempio, src/app o una sottocartella dedicata)
2. Seleziona **New** dal menu contestuale
3. Dal sottomenu, scegli **Angular Schematic**....
4. Nella finestra che si apre, seleziona **Component** dall'elenco degli schemi disponibili
5. Inserisci il **nome del componente** nel campo apposito e personalizza eventuali opzioni (come la creazione di un componente standalone, se necessario)
6. Conferma la creazione cliccando su **OK**

2.2 Directives

Le **Directives** in Angular sono istruzioni speciali che possono essere aggiunte agli elementi del DOM per cambiare il loro aspetto, comportamento o struttura. Le direttive estendono le funzionalità del framework, consentendo di creare elementi dinamici e interattivi.

In Angular, esistono tre tipi principali di direttive:

1. **Directives Strutturali:** Le direttive strutturali cambiano la struttura del DOM, aggiungendo o rimuovendo elementi in base a determinate condizioni.

   Le direttive strutturali più comuni sono:

   - ***ngIf:** Mostra o nasconde un elemento in base a una condizione
   - ***ngFor:** Crea un elenco di elementi ripetendo un template per ciascun elemento di un array:

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

*Figura 1:Qui, *ngFor genera un elemento <li> per ogni elemento presente nell'array*

- **Directives Attributive:** Le direttive attributive cambiano l'aspetto o il comportamento di un elemento senza modificare la struttura del DOM. Alcune direttive attributive comuni includono:
    - **ngStyle:** Applica stili CSS dinamici a un elemento in base a una condizione:
    - **ngClass**: Aggiunge o rimuove classi CSS su un elemento in base a una condizione.

2.3 Data Binding

Il **data binding** in Angular è un meccanismo che collega i dati della logica del componente con l'interfaccia utente (template HTML) dell'applicazione, permettendo di sincronizzare e gestire in modo dinamico i contenuti mostrati all'utente. Grazie al data binding, è possibile visualizzare, aggiornare e gestire i dati senza dover modificare manualmente il DOM.

Angular offre quattro modalità principali di data binding:

- **Two-Way Binding:** Il two-way binding consente di sincronizzare i dati tra la classe e il template. Questo significa che quando l'utente modifica il valore nel template, il modello (cioè la variabile nel componente) viene aggiornato automaticamente, e viceversa. Per il two-way binding, Angular utilizza la direttiva ngModel, disponibile dal modulo FormsModule;
- **Interpolation (Interpolazione):** L'interpolazione consente di inserire valori direttamente nel template usando le doppie parentesi graffe {{ }}. Questo tipo di binding permette di visualizzare il valore di una proprietà della classe TypeScript all'interno del template HTML;
- **Property Binding:** Il property binding consente di collegare una proprietà dell'elemento HTML a una variabile del componente, utilizzando la sintassi [property]="valore". Questo metodo è utile quando si desidera impostare dinamicamente attributi HTML, come src di un'immagine o disabled di un pulsante;
- **Event Binding:** L'event binding permette di ascoltare eventi (come click, keyup, mouseover) e di eseguire una funzione del componente in risposta all'evento. La sintassi è (evento)="metodo()".

Tipo	Sintassi	Descrizione
Interpolation	{{property}}	Inserisce direttamente il valore di una proprietà.
Property Binding	[property]="value"	Collega una proprietà HTML a una variabile del componente.
Event Binding	(event)="method()"	Ascolta un evento HTML e chiama un metodo del componente.
Two-Way Binding	[(ngModel)]="property"	Sincronizza i dati tra il modello e il template.

2.4 Pipes

In Angular, i **pipes** sono strumenti utili per trasformare i dati all'interno del template prima che vengano visualizzati. Essi consentono di formattare e manipolare rapidamente i valori (come stringhe, date e numeri) direttamente nel template HTML, senza bisogno di modificare la logica del componente. I pipes semplificano il codice, rendendolo più leggibile e mantenendo separata la logica di formattazione dai dati. I pipes in Angular sono applicati utilizzando il simbolo | (pipe) nel template, seguito dal nome del pipe e, se necessario, dai parametri. {{ valore | nomePipe: parametro1 : parametro2 }}. Qui, valore è il dato da trasformare, nomePipe è il nome del pipe che esegue la trasformazione e i parametri sono valori opzionali che specificano ulteriormente la formattazione. Angular fornisce una serie di pipes integrati che coprono molte delle necessità comuni di formattazione:

- **Date Pipe (date):** Formata le date in diversi modi. Il pipe date accetta un parametro che specifica il formato della data:

```
<p>{{ oggi | date:'dd/MM/yyyy' }}</p>
```

Figura 2:Se oggi contiene una data, il risultato sarà mostrato nel formato giorno/mese/anno (ad esempio, 14/11/2024)

- **Uppercase e Lowercase Pipes (uppercase, lowercase)**: Converti una stringa in maiuscolo o minuscolo;
- **Currency Pipe (currency)**: Formatizza un numero come valuta, con l'opzione di specificare il simbolo e il numero di decimali;

- **Decimal Pipe (number)**: Formatizza i numeri in un formato decimale specifico;
- **Percent Pipe (percent)**: Visualizza i numeri come percentuali.

È possibile applicare più pipes su uno stesso valore, combinandoli per ottenere il risultato desiderato. Questo processo si chiama **chaining** (concatenazione) e si realizza semplicemente aggiungendo un altro | seguito dal nome del pipe successivo.

```
<p>{{ 'hello world' | uppercase | lowercase }}</p>
```

Se i pipes integrati non sono sufficienti, Angular consente di creare pipes personalizzati. Questo è utile quando si ha bisogno di una trasformazione specifica per il proprio progetto.

3 Routing in Angular

3.1 Rounting in Angular

Il **routing** in Angular è un sistema che consente di navigare tra diverse pagine o sezioni di un'applicazione senza dover ricaricare l'intera pagina, migliorando l'esperienza utente. Questo sistema permette di gestire percorsi URL specifici, collegando ogni percorso a un componente.
Il routing è configurato mediante il modulo RouterModule, che gestisce i percorsi dell'applicazione. Definire le rotte significa collegare i percorsi URL a componenti specifici, in modo che quando un utente visita un determinato URL, il componente associato viene visualizzato automaticamente.
Per abilitare il routing in un'app Angular, è necessario configurare RouterModule e definire le rotte principali. Ecco i passaggi:

- **Definire le Rotte**: Le rotte vengono configurate come un array di oggetti Routes, in cui ogni oggetto rappresenta una singola rotta. Ogni rotta ha una proprietà path (l'URL) e una component (il componente da visualizzare):

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];
```

- **Configurare il RouterModule**: Una volta definite le rotte, bisogna configurarle nel modulo principale dell'applicazione:

```
import { Routes, provideRouter } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];

export const appRouting = provideRouter(routes);
```

- **Visualizzare le Rotte:**Utilizza <router-outlet> nel template principale (app.component.html) come punto di inserimento dinamico per i componenti:

```
<nav>
  <a routerLink="/home">Home</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

3.2 Navigazione Programmatica

In Angular, oltre alla navigazione tramite collegamenti statici (come i routerLink negli elementi <a>), è possibile gestire la navigazione in modo programmatico utilizzando il servizio **Router**. Questo è utile in situazioni in cui la navigazione dipende da condizioni dinamiche o logiche specifiche del componente, come l'autenticazione, il salvataggio dei dati, o la gestione di eventi.

Per poter utilizzare il servizio Router, dobbiamo prima di tutto importarlo e iniettarlo nel componente. Una volta fatto, possiamo invocare il metodo navigate() per spostare l'utente verso un altro percorso all'interno dell'applicazione.

Esempio di Navigazione Programmatica: Supponiamo di voler reindirizzare l'utente alla pagina del dashboard dopo il login. Per farlo, importiamo il servizio Router dal modulo @angular/router e lo inseriamo nel costruttore della classe del componente, come mostrato nell'esempio:

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

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html'
})
export class LoginComponent {
  constructor(private router: Router) {}

  navigateToDashboard() {
    this.router.navigate(['/dashboard']);
  }
}
```

Figura 3:In questo caso, il metodo navigateToDashboard() utilizza this.router.navigate(['/dashboard']) per reindirizzare l'utente alla pagina del dashboard.

Spesso è necessario passare informazioni specifiche alla nuova pagina, come l'ID di un prodotto o di un utente. Possiamo farlo aggiungendo parametri alla rotta. Ad esempio, per navigare alla pagina di un prodotto specifico, possiamo usare:

```
this.router.navigate(['/product', productId]);
```

Figura 4: Qui, productId rappresenta un parametro che viene incorporato nell'URL /product/{productId}, in modo che il componente di destinazione possa accedere al valore di questo parametro e visualizzare i dettagli corretti.

Oltre ai parametri di rotta, Angular consente di utilizzare i **query params**, che sono parametri aggiunti all'URL dopo il punto interrogativo (?). I query params sono utili quando abbiamo bisogno di passare

dati meno rilevanti per la struttura dell'applicazione, come opzioni di filtraggio o di ordinamento.

Possiamo impostare i query params utilizzando la proprietà queryParams:

```
this.router.navigate(['/search'], { queryParams: { query: 'angular', page: 1 } }
```

Figura 5:Questo comando naviga alla pagina /search, aggiungendo query=angular&page=1 all'URL finale (/search?query=angular&page=1).

La navigazione può essere effettuata anche in modo **relativo** rispetto al percorso attuale. Per fare questo, usiamo la proprietà relativeTo, che permette di indicare come base la rotta corrente:

```
this.router.navigate(['details'], { relativeTo: this.route });
```

Figura 6:In questo caso, details sarà aggiunto all'URL corrente, risultando ad esempio in /currentRoute/details.

Angular offre anche ulteriori opzioni per configurare la navigazione, come l'uso di **frammenti** (fragments), che consentono di far scorrere la pagina verso una specifica sezione, o opzioni di gestione del comportamento di scorrimento della pagina.

La navigazione programmatica è particolarmente utile quando si vuole:

- **Gestire flussi dinamici:** come la navigazione condizionale basata sull'esito di un'azione, come un form completato o l'autenticazione.
- **Reindirizzare automaticamente:** dopo una determinata azione, come il login o il logout, o in caso di mancanza di autorizzazione.
- **Passare dati dinamici:** attraverso parametri di rotta o query params, in modo da rendere la navigazione più flessibile e adatta a scenari specifici.

3.3 Guards (Route Protection)

In Angular, i **guards** sono strumenti che aiutano a proteggere le rotte dell'applicazione, regolando l'accesso a determinate pagine in base a condizioni specifiche. Ad esempio, un guard può controllare se un utente è autenticato prima di consentire l'accesso a una pagina privata o protetta.

I guards agiscono come "guardiani" delle rotte e permettono di implementare logiche personalizzate che determinano se una rotta può essere attivata, disattivata, caricata o uscita. Questo meccanismo è particolarmente utile per creare applicazioni sicure e per gestire flussi condizionali.

Angular offre diversi tipi di guards, ognuno con un ruolo specifico:

- **CanActivate:** Controlla se una rotta può essere attivata. È utile per gestire l'accesso a una pagina protetta, come la dashboard di un utente autenticato;

- **CanActivateChild:** Simile a CanActivate, ma applicato alle rotte figlie. Controlla se l'utente può accedere alle sottorotte di una rotta protetta;
- **CanDeactivate:** Controlla se è possibile uscire da una rotta. Viene spesso usato per avvisare l'utente quando sta per lasciare una pagina con dati non salvati;
- **Resolve:** Recupera i dati necessari prima di attivare una rotta, garantendo che i dati siano disponibili al caricamento della pagina;
- **CanLoad:** Controlla se un modulo di rotta può essere caricato. Questo è utile per impedire il caricamento di moduli specifici, come quelli che richiedono autenticazione.

Per creare una nuova guards in IntelliJ IDEA, segui questi passaggi:

1. Fai clic destro sulla cartella in cui desideri creare il componente (ad esempio, src/app o una sottocartella dedicata)
2. Seleziona **New** dal menu contestuale
3. Dal sottomenu, scegli **Angular Schematic**....
4. Nella finestra che si apre, seleziona **Guards** dall'elenco degli schemi disponibili
5. Inserisci il **nome** nel campo apposito e personalizza eventuali opzioni (come la creazione di un componente standalone, se necessario)
6. Conferma la creazione cliccando su **OK**

Supponiamo di voler creare un guard AuthGuard per impedire l'accesso a utenti non autenticati.

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

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(): boolean {
    if (this.authService.isLoggedIn()) {
      return true;
    } else {
      this.router.navigate(['/login']);
      return false;
    }
  }
}
```

In questo esempio:

- AuthGuard verifica se l'utente è autenticato chiamando il metodo isLoggedIn() di AuthService;
- Se l'utente è autenticato, il guard restituisce true e consente l'accesso;

- Se l'utente non è autenticato, il guard reindirizza alla pagina di login e restituisce false.

Per utilizzare un guard, bisogna applicarlo alla configurazione della rotta.

Si utilizza la proprietà **canActivate** (o un'altra a seconda del tipo di guard) nella definizione della rotta:

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

const routes: Routes = [
  { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard]
  { path: 'profile', component: ProfileComponent, canActivate: [AuthGuard] }
];
```

4 Gestione delle Entità in Angular

4.1 Definizione di Entità

Le **entità** in Angular rappresentano i dati centrali di un'applicazione, come utenti, prodotti, ordini o qualsiasi altra informazione principale. La gestione delle entità include la definizione di modelli di dati, che strutturano i dati in modo uniforme e organizzato, e l'interazione con i servizi per recuperare, aggiornare e sincronizzare i dati con fonti esterne, come un server.

Un'entità rappresenta un oggetto di dati specifico per il dominio dell'applicazione, descrivendo le informazioni necessarie per definire un concetto. Per esempio, in un'app di e-commerce, le entità possono essere Product, User, Order, ciascuna contenente proprietà che caratterizzano quell'entità.

Le entità vengono definite come classi di modelli in TypeScript, che organizzano e validano i dati in modo coerente, facilitando l'integrazione con servizi e componenti dell'applicazione. La classe di un'entità specifica le proprietà e il tipo di dati di ciascuna di esse, garantendo che i dati gestiti siano strutturati correttamente:

```
export class Product {
  id: number;
  name: string;
  price: number;
  description?: string;

  constructor(id: number, name: string, price: number, description?: string)
    this.id = id;
    this.name = name;
    this.price = price;
    this.description = description;
  }
}
```

Figura 7:In questo esempio, l'entità Product è definita con proprietà come id, name, price e description. Utilizzando una classe di modello, possiamo controllare la struttura e i tipi di questi dati.

4.2 Creazione di Modelli di Dati

La creazione dei modelli di dati è un passaggio fondamentale per rappresentare le entità all'interno dell'applicazione. I modelli di dati sono classi TypeScript che definiscono le proprietà principali di

un'entità e il loro tipo. Questi modelli vengono poi utilizzati nei componenti e nei servizi per manipolare i dati in modo organizzato. Vantaggi dei **Modelli di Dati:**

- **Controllo dei Tipi:** Garantisce che le proprietà abbiano i tipi corretti, prevenendo errori di runtime.
- **Riutilizzabilità**: Un modello può essere riutilizzato in vari componenti e servizi, riducendo la duplicazione del codice.
- **Integrazione con i Servizi:** I modelli permettono di gestire facilmente le entità ricevute dalle API o database esterni.

Per creare una nuova entità in IntelliJ IDEA, segui questi passaggi:

1. Fai clic destro sulla cartella in cui desideri creare il componente (ad esempio, src/app o una sottocartella dedicata).
2. Seleziona **New** dal menu contestuale.
3. Dal sottomenu, scegli **Angular Schematic**....
4. Nella finestra che si apre, seleziona **Class** dall'elenco degli schemi disponibili.
5. Inserisci il **nome** nel campo apposito.
6. Conferma la creazione cliccando su **OK.**

4.3 Interazione con i servizi

Per gestire i dati delle entità, Angular utilizza i **servizi**. I servizi permettono di centralizzare la logica di accesso e manipolazione dei dati, facilitando il recupero e l'aggiornamento dei dati tramite chiamate API verso un server esterno.

5 Servizi in Angular

5.1 Ruolo dei servizi

I **servizi** in Angular sono componenti dedicati a eseguire compiti specifici e condivisi, come il recupero di dati da un server, la gestione delle notifiche o la manipolazione dei dati in generale. Utilizzare i servizi consente di separare la logica dell'applicazione dalla logica di visualizzazione nei componenti, mantenendo il codice organizzato e modulare.

Il ruolo principale dei servizi è fornire **funzionalità riutilizzabili** e condivisibili tra vari componenti di un'applicazione. I servizi contengono la logica di business dell'app, consentendo ai componenti di rimanere snelli e concentrati solo sull'interfaccia utente. Ecco alcuni esempi di compiti che i servizi gestiscono comunemente:

- **Accesso ai Dati**: I servizi possono gestire chiamate HTTP per recuperare, aggiornare o eliminare dati da un server remoto.
- **Condivisione dei Dati**: Possono mantenere dati o stati condivisi tra più componenti.
- **Esecuzione di Logica Complessa**: I servizi possono elaborare dati o applicare algoritmi complessi che non appartengono ai singoli componenti.
- **Gestione delle Notifiche**: Gestiscono notifiche e messaggi per l'interazione utente.

5.2 Creazione di un Servizio

Per creare un nuovo servizio in IntelliJ IDEA, segui questi passaggi:

1. Fai clic destro sulla cartella in cui desideri creare il componente (ad esempio, src/app o una sottocartella dedicata).
2. Seleziona **New** dal menu contestuale.
3. Dal sottomenu, scegli **Angular Schematic**....
4. Nella finestra che si apre, seleziona **Service** dall'elenco degli schemi disponibili.
5. Inserisci il **nome** nel campo apposito.
6. Conferma la creazione cliccando su **OK.**

Questo comando genera un file di servizio TypeScript nella cartella specificata e registra automaticamente il servizio con il decoratore @Injectable, rendendolo disponibile per l'injection in tutta l'app:

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

@Injectable({
  providedIn: 'root'
})
export class LogService {
  log(messaggio: string) {
    console.log('Log:', messaggio);
  }
}
```

Figura 8:In questo esempio, LogService è un servizio semplice che fornisce un metodo log() per visualizzare messaggi di log nella console. Grazie alla proprietà providedIn: 'root', il servizio viene iniettato nel modulo radice, rendendolo disponibile globalmente

5.3 Utilizzo dei Servizi con Dependency Injection (DI)

In Angular, i servizi vengono iniettati nei componenti o in altri servizi tramite **Dependency Injection (DI)**. DI è un sistema che permette ad Angular di gestire automaticamente le dipendenze, instanziando e passando i servizi ai componenti che ne hanno bisogno.

Iniettare un Servizio in un Componente: Per usare un servizio in un componente, è necessario dichiararlo come dipendenza nel costruttore del componente. Angular si occupa di creare l'istanza del servizio e di iniettarla automaticamente:

```
import { Component, OnInit } from '@angular/core';
import { LogService } from './services/log.service';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html'
})
export class DashboardComponent implements OnInit {
  constructor(private logService: LogService) {}

  ngOnInit(): void {
    this.logService.log('Dashboard inizializzato');
  }
}
```

Figura 9: In questo esempio, LogService viene iniettato nel componente DashboardComponent, che lo utilizza per registrare un messaggio di log durante l'inizializzazione.

5.4 Servizi per l'Accesso ai Dati

Uno degli usi più comuni dei servizi in Angular è l'accesso ai dati. Angular utilizza HttpClient, un modulo che consente di eseguire chiamate HTTP per comunicare con server e API esterne in modo semplice ed efficiente. Per integrare HttpClient per l'accesso ai dati, è possibile importare HttpClientModule direttamente nel componente standalone in cui viene utilizzato. Di seguito è riportato un esempio di configurazione per integrare HttpClient in un componente standalone:

```
import { HttpClientModule } from '@angular/common/http';
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { inject } from '@angular/core';

@Component({
  standalone: true,
  imports: [HttpClientModule],
  selector: 'app-product-list',
  templateUrl: './product-list.component.html'
})
export class ProductListComponent {
  private http = inject(HttpClient);

  constructor() {
    this.http.get('https://api.example.com/products').subscribe(data => {
      console.log(data);
    });
  }
}
```

6 Observable e RxJS

6.1 Introduzione a RxJS e Observable

In Angular, gli **Observable** e la libreria **RxJS** (Reactive Extensions for JavaScript) sono fondamentali per gestire operazioni asincrone, come il recupero di dati da un server, l'elaborazione di eventi o l'aggiornamento in tempo reale di contenuti. Gli Observable permettono di monitorare e reagire a flussi di dati in arrivo in modo reattivo e sono particolarmente utili per applicazioni che richiedono interattività e aggiornamenti costanti.

RxJS è una libreria basata sul paradigma della programmazione reattiva, che consente di creare, trasformare e combinare flussi di dati asincroni tramite **Observable**. Gli Observable rappresentano sequenze di dati che possono essere osservate, o "sottoscritte", permettendo ai componenti di reagire in tempo reale a nuovi valori. In Angular, HttpClient e molti altri servizi utilizzano Observable per restituire i dati.

Cos'è un Observable? Un Observable è un oggetto che emette valori nel tempo, come una lista di eventi o dati. Ad esempio, può emettere un singolo valore (come una risposta HTTP) o più valori (come un flusso di eventi da un input utente). Ogni Observable può essere **sottoscritto** per monitorare i suoi valori ed eseguire operazioni quando vengono emessi.

6.2 Creazione e Utilizzo degli Observable

Gli Observable possono essere creati e utilizzati con RxJS per gestire flussi di dati. Esistono diversi modi per creare Observable, inclusi i metodi create e of, ma una delle modalità più comuni in Angular è quella di sfruttare HttpClient:

```
import { of } from 'rxjs';

const observable$ = of(1, 2, 3);
observable$.subscribe(value => console.log(value));
```

Figura 10:Qui, of crea un Observable che emette i valori 1, 2, 3 in sequenza.

Con HttpClient, possiamo recuperare dati da un server e ottenere un Observable:

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Product } from './models/product';

constructor(private http: HttpClient) {}

getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>('https://api.example.com/products');
}
```

6.3 Operatori RxJS

Gli operatori RxJS consentono di trasformare, filtrare, combinare e gestire i flussi di dati degli Observable. Esistono numerosi operatori in RxJS, suddivisi in diverse categorie (es., operatori di trasformazione, operatori di combinazione e operatori di filtraggio).

Alcuni Operatori Comuni:

- **map:** Trasforma i valori emessi da un Observable:

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

const squared$ = observable$.pipe(
  map(value => value * value)
);
squared$.subscribe(value => console.log(value));
```

Figura 11: In questo esempio, l'operatore map eleva al quadrato ogni valore emesso.

- **filter**: Filtra i valori in base a una condizione:

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

const even$ = observable$.pipe(
  filter(value => value % 2 === 0)
);
even$.subscribe(value => console.log(value));
```

Figura 12: Qui, filter lascia passare solo i numeri pari.