Guida Rapida ad Angular (Dark Edition)

November 25, 2025

INDICE

1 Introduzione ad Angular

Angular è un framework front-end sviluppato da Google per creare applicazioni web single-page (SPA) moderne, modulari e scalabili. Basato su TypeScript, Angular utilizza un'architettura a componenti e servizi, promuove la separazione delle responsabilità e semplifica la gestione dello stato, delle richieste HTTP e della navigazione.

1.1 Vantaggi principali di Angular

- **Component-based architecture**: ogni parte dell'interfaccia è un componente riutilizzabile.
- **Two-way data binding**: sincronizzazione automatica tra modello e view.
- **Dependency Injection**: semplifica l'iniezione di servizi nei componenti.
- **TypeScript**: forte tipizzazione e strumenti di debugging.
- **CLI ufficiale**: scaffolding rapido di progetti, componenti, servizi, moduli e test.
- **Routing integrato**: gestione delle route e navigazione tra view.

1.2 Avviare un progetto Angular

1. Installare Node.js e Angular CLI:

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

2. Creare un nuovo progetto:

   ```
   ng new nome-progetto
   ```

3. Avviare l'app in sviluppo:

   ```
   cd nome-progetto
   ng serve
   ```

   L'app sarà accessibile su `http://localhost:4200`.

4. Struttura generata dalla CLI:
   - `src/app`: cartella principale per componenti, servizi e moduli.
   - `app.component.ts/html/css`: componente root.
   - `app.module.ts`: modulo principale che importa componenti e servizi.

1.3 Concetti fondamentali

- **Componenti**: classi TypeScript con template HTML e stili CSS. Gestiscono la logica di una parte dell'interfaccia.
- **Servizi**: classi per logica condivisa, chiamate HTTP, gestione dello stato o interazione con storage locale.
- **Moduli**: raggruppano componenti e servizi correlati. Ogni progetto ha un modulo root (`AppModule`) e può avere moduli feature.
- **Directive**: estendono il comportamento di elementi HTML.
- **Pipe**: trasformano i dati nel template (es: formattazione date o numeri).
- **Routing**: definizione delle rotte nel modulo di routing per navigare tra componenti.
- **HTTP Interceptor**: intercetta richieste HTTP per aggiungere token, gestire errori o logging.
- **LocalStorage / SessionStorage**: memorizzazione locale di dati persistenti.

1.4 Struttura di un componente Angular

```
@Component({
  selector: 'app-nome',
  templateUrl: './nome.component.html',
  styleUrls: ['./nome.component.css']
})
export class NomeComponent {
  // proprietà e metodi del componente
}
```

1.5 Servizio base

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

  getData(): Observable<Data[]> {
    return this.http.get<Data[]>('/api/data');
  }
}
```

1.6 Interceptor base

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = localStorage.getItem('token');
    const cloned = req.clone({
      setHeaders: { Authorization: `Bearer ${token}` }
    });
    return next.handle(cloned);
  }
}
```

2 Gestione dei Servizi Angular

In Angular, i servizi sono fondamentali per separare la logica di business dai componenti, gestire chiamate HTTP, e interagire con API esterne o storage locale. I servizi permettono di centralizzare la logica condivisa e sfruttare la Dependency Injection.

2.1 Servizi base

Un servizio base può essere creato tramite CLI o manualmente. I servizi vengono dichiarati con il decoratore `@Injectable` e registrati nel modulo root o in moduli feature.

```
@Injectable({
  providedIn: 'root'  // disponibile in tutta l'app
})
```

```
export class NomeService {
  constructor(private http: HttpClient) {}

  getData(): Observable<Data[]> {
    return this.http.get<Data[]>('/api/data');
  }
}
```

2.2 Integrazione con API esterne

Per consumare API esterne (REST o Firebase) si usa il servizio Angular con `HttpClient`.
Esempio: chiamata a un endpoint Firebase Realtime Database:

```
@Injectable({ providedIn: 'root' })
export class FirebaseService {
  private baseUrl = 'https://<PROJECT_ID>.firebaseio.com';

  constructor(private http: HttpClient) {}

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>(`${this.baseUrl}/users.json`);
  }

  addUser(user: User): Observable<User> {
    return this.http.post<User>(`${this.baseUrl}/users.json`, user);
  }
}
```

2.3 Gestione dei servizi autonomi

Alcuni servizi possono essere completamente indipendenti, senza dipendenze da HTTP, per gestire stato interno, logica, calcoli o interazioni con LocalStorage/SessionStorage:

```
@Injectable({ providedIn: 'root' })
export class StateService {
  private currentUser: User | null = null;

  setUser(user: User) {
    this.currentUser = user;
    localStorage.setItem('user', JSON.stringify(user));
  }

  getUser(): User | null {
    const stored = localStorage.getItem('user');
    return stored ? JSON.parse(stored) : this.currentUser;
  }

  clearUser() {
    this.currentUser = null;
    localStorage.removeItem('user');
  }
}
```

2.4 Servizi con Dependency Injection multipla

È possibile iniettare più servizi in un componente per orchestrare logica complessa:

```
@Component({...})
export class DashboardComponent {
  constructor(
    private firebaseService: FirebaseService,
    private stateService: StateService
  ) {}

  loadData() {
    this.firebaseService.getUsers().subscribe(users => {
      console.log(users);
      this.stateService.setUser(users[0]);
    });
  }
}
```

2.5 Best practice nella gestione dei servizi

- Usare `@Injectable( providedIn: 'root' )` per servizi globali.
- Separare la logica di business dai componenti.
- Non fare chiamate HTTP direttamente nei componenti.
- Gestire gli errori con operatori RxJS (`catchError`, `retry`).
- Usare Interceptor per aggiungere token e logging a tutte le richieste HTTP.
- Documentare i servizi e le API integrate.

3 Data Binding, Directives e Forms in Angular

3.1 Tipi di Data Binding

Angular offre diversi modi per collegare il template HTML alla logica TypeScript dei componenti:

- **Interpolation ( )**: visualizza valori del componente nel template.

  ```
  <p>{{ title }}</p> <!-- Mostra il valore di title in HTML -->
  ```

  ‘‘‘

- **Property Binding [prop]**: lega proprietà HTML a variabili TS.

  ‘‘‘

  ```
  <img [src]="imageUrl" alt="Immagine dinamica">
  ```

  ‘‘‘

- **Event Binding (event)**: cattura eventi HTML.

  ‘‘‘

  ```
  <button (click)="onClick()">Cliccami</button>
  ```

```

- **Two-way Binding [(ngModel)]**: sincronizza variabile TS e input HTML.

```
<input [(ngModel)]="username" placeholder="Nome utente">

Ciao {{ username }}
```

## 3.2 Directives Strutturali

Le directives strutturali modificano la struttura del DOM:

- **\*ngIf**: mostra o nasconde elementi.

```
<p *ngIf="isLoggedIn">Benvenuto!
```

```

- ***ngFor**: cicla su array o oggetti.

```

  <li *ngFor="let item of items; let i = index">
    {{ i+1 }} - {{ item.name }}

```

```

- **\*ngSwitch**: scelta tra più template.

```
<div [ngSwitch]="role">
  <p *ngSwitchCase="'admin'">Admin view
  <p *ngSwitchCase="'user'">User view
  <p *ngSwitchDefault>Guest view

```

## 3.3 Directives Attributo

Le directives attributo modificano comportamento o stile di un elemento:

- **[ngClass]**: classi CSS dinamiche

```
<div [ngClass]="{'active': isActive, 'disabled': isDisabled}">
  Contenuto

```

```

- **[ngStyle]**: stili inline dinamici

  ‘‘‘

  ```
  <p [ngStyle]="{’color’: isError ? ’red’ : ’green’}">
    Messaggio di stato
  </p>
  ```

3.4 Forms in Angular

3.4.1 Template-driven Forms

```
<form #userForm="ngForm" (ngSubmit)="onSubmit(userForm)">
  <input name="username" ngModel required>
  <input type="email" name="email" ngModel>
  <button type="submit">Invia</button>
</form>
```

Note: Utilizzabili per form semplici, validazioni base con attributi HTML5 + `ngModel`.

3.4.2 Reactive Forms

```
import { FormGroup, FormControl, Validators } from ’@angular/forms’;

export class AppComponent {
userForm = new FormGroup({
username: new FormControl(’’, [Validators.required, Validators.minLength(3)]),
email: new FormControl(’’, [Validators.email])
});

onSubmit() {
console.log(this.userForm.value);
}
}
```

```
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
  <input formControlName="username">
  <input formControlName="email">
  <button type="submit">Invia</button>
</form>
```

3.5 Pipes

I pipes trasformano i dati direttamente nel template:

- **Pipe built-in**

  ```
  <p>{{ today | date:’shortDate’ }}</p>
  <p>{{ name | uppercase }}</p>
  <p>{{ price | currency:’EUR’ }}</p>
  ```

  “‘

- **Pipe personalizzate**

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'exclaim'})
export class ExclaimPipe implements PipeTransform {
transform(value: string): string {
return value + '!!!';
}
}

<p>{{ 'Ciao' | exclaim }}</p> <!-- Risultato: Ciao!!! -->
```

3.6 Eventi e metodi nei componenti

```
@Component({...})
export class MyComponent {
username = '';

onInput(event: any) {
this.username = event.target.value;
}

onClick() {
alert('Hai cliccato!');
}
}

<input (input)="onInput($event)">
<button (click)="onClick()">Click</button>

<p>Ciao {{ username }}</p>
```

3.7 Best Practices

- Usare **Reactive Forms** per form complessi con logica di validazione avanzata.
- Tenere la logica dei componenti leggera e delegare servizi per chiamate HTTP.
- Riutilizzare **componenti** e **pipes** per modularità.
- Usare **AsyncPipe** per Observable e Promise.
- Preferire **trackBy** con *ngFor per performance.

4 Lifecycle Hooks dei Componenti Angular

Angular fornisce una serie di metodi chiamati **lifecycle hooks** che permettono di agganciarsi a eventi chiave del ciclo di vita dei componenti e delle direttive. Questi hook consentono di eseguire logica custom in momenti precisi: creazione, aggiornamento, distruzione e cambiamenti di input.

4.1 ngOnChanges

- Triggerato quando cambia il valore di un input binding (@Input) nel componente.

- Permette di reagire a cambiamenti dei dati esterni.

```
export class MyComponent implements OnChanges {
@Input() data: string;

ngOnChanges(changes: SimpleChanges) {
console.log('Cambiamento input:', changes.data.currentValue);
}
}
```

4.2 ngOnInit

- Chiamato una sola volta subito dopo la prima inizializzazione degli input del componente.
- Ideale per inizializzazioni di dati o chiamate HTTP tramite servizi.

```
export class MyComponent implements OnInit {
ngOnInit() {
console.log('Componente inizializzato');
this.loadData();
}

loadData() {
// chiamata a un servizio
}
}
```

4.3 ngDoCheck

- Chiamato ad ogni rilevamento dei cambiamenti (change detection) del componente.
- Permette di implementare controlli custom oltre al default di Angular.

```
ngDoCheck() {
console.log('Change detection attiva');
}
```

4.4 ngAfterContentInit

- Chiamato dopo che Angular ha proiettato il contenuto esterno nel componente (<ng-content>).

4.5 ngAfterContentChecked

- Triggerato dopo ogni ciclo di change detection del contenuto proiettato.

4.6 ngAfterViewInit

- Chiamato dopo che Angular ha inizializzato le view e le view figlie del componente.
- Utile per accedere a template reference variables o elementi DOM via @ViewChild.

```
export class MyComponent implements AfterViewInit {
@ViewChild('myDiv') myDiv: ElementRef;

ngAfterViewInit() {
console.log(this.myDiv.nativeElement.innerHTML);
}
}
```

4.7 ngAfterViewChecked

- Triggerato dopo ogni ciclo di change detection della view del componente e delle view figlie.

4.8 ngOnDestroy

- Chiamato subito prima che il componente venga distrutto.
- Utile per pulire subscription, timers o listener per evitare memory leaks.

```
export class MyComponent implements OnDestroy {
subscription: Subscription;

ngOnDestroy() {
this.subscription.unsubscribe();
console.log('Componente distrutto');
}
}
```

4.9 Riepilogo Lifecycle Hook

Hook Angular	**Descrizione**
ngOnChanges	Cambiamento input (@Input)
ngOnInit	Dopo la prima inizializzazione
ngDoCheck	Ad ogni change detection
ngAfterContentInit	Dopo la proiezione del contenuto
ngAfterContentChecked	Dopo ogni change detection del contenuto
ngAfterViewInit	Dopo l'inizializzazione della view
ngAfterViewChecked	Dopo ogni change detection della view
ngOnDestroy	Prima della distruzione del componente

Table 1: Lifecycle hooks principali di Angular

4.10 Best Practices

- Utilizzare `ngOnInit` per inizializzazioni di dati asincroni.
- Evitare logica complessa in `ngDoCheck` per non degradare le performance.
- Pulire subscription e listener in `ngOnDestroy`.
- Non manipolare direttamente il DOM, preferire `@ViewChild` o direttive.

5 Gestione dei Form in Angular

I form in Angular sono essenziali per raccogliere dati dagli utenti e inviarli a backend o servizi. Angular offre due approcci principali:

5.1 Template-driven Forms

- Basati sul template HTML: la logica del form è gestita direttamente nel template.
- Facili da implementare per form semplici.
- Utilizzano le direttive di Angular come `ngModel`, `ngForm`, `required`, ecc.
- Esempio base:

```
<form #form="ngForm" (ngSubmit)="onSubmit(form)">
  <input type="text" name="username" ngModel required>
  <input type="email" name="email" ngModel>
  <button type="submit">Invia</button>
</form>
```

- In questo approccio, i dati del form si trovano nell'oggetto `form.value`.
- Validazioni semplici tramite attributi HTML (`required`, `minlength`, ecc.).

5.2 Reactive Forms (o Model-driven Forms)

- Basati su TypeScript: il form è definito in codice e non solo in template.
- Più potenti e adatti a form complessi.
- Utilizzano `FormGroup`, `FormControl` e `FormBuilder`.
- Permettono validazioni dinamiche, custom validator e gestione avanzata dello stato.
- Esempio base:

```
this.userForm = this.fb.group({
  username: ['', [Validators.required, Validators.minLength(3)]],
  email: ['', [Validators.email]]
});

onSubmit() {
  console.log(this.userForm.value);
}
```

- Nel template:

```
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
  <input formControlName="username">
  <input formControlName="email">
  <button type="submit">Invia</button>
</form>
```

5.3 Validazioni e feedback utente

- Validazioni integrate: `required`, `minlength`, `maxlength`, `pattern`, `email`.
- Validazioni custom: funzioni che ritornano `null` se valido o un oggetto errore se non valido.
- Mostrare messaggi di errore condizionali con `*ngIf`:

```
<div *ngIf="userForm.controls['username'].invalid && userForm.controls['username'].touched">
  Username obbligatorio
</div>
```

5.4 Passaggio di dati tra componenti

- **Parent →Child**: usare `@Input` per passare valori dal componente genitore al figlio.
- **Child →Parent**: usare `@Output` con `EventEmitter` per inviare dati dal figlio al genitore.
- **Componenti non correlati**: usare servizi con `BehaviorSubject` o `Subject` per comunicazione tramite Observable.
- Esempio `@Input / @Output`:

```
// child.component.ts
@Input() userData: any;
@Output() submitData = new EventEmitter<any>();

onSubmit() {
  this.submitData.emit(this.userData);
}

// parent.component.html
<app-child [userData]="currentUser" (submitData)="handleSubmit($event)"></app-child>
```

5.5 Invio dati a backend o servizi esterni

- Creare un servizio con `HttpClient` per comunicare con API REST o Firebase.
- Esempio invio dati:

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

  sendForm(data: any) {
    return this.http.post('/api/form', data);
  }
}
```

- Chiamare il servizio dal componente:

```
onSubmit() {
  this.apiService.sendForm(this.userForm.value)
    .subscribe(response => console.log(response));
}
```

- Gestione asincrona con `Observable` e `subscribe`, oppure con `async pipe` nel template.

5.6 Gestione dello stato dei form

- Stato dei form: `valid`, `invalid`, `dirty`, `pristine`, `touched`, `untouched`.
- Permette di abilitare/disabilitare pulsanti o mostrare feedback.
- Esempio:

```
<button type="submit" [disabled]="userForm.invalid">Invia</button>
</verbatim}
\end{itemize}

\subsection{Integrazione con Firebase o API esterne}

\begin{itemize}
    \item Usare servizi Angular per connettersi a Firebase Realtime Database o Firestore.
    \item Esempio invio dati a Firestore:

\begin{verbatim}
constructor(private afs: AngularFirestore) {}

saveUser(user: any) {
  return this.afs.collection('users').add(user);
}
```

- Per API REST esterne: usare `HttpClient` con header e token se necessario.

6 Routing Avanzato in Angular

Il routing in Angular permette di gestire la navigazione tra componenti e view in modo modulare e scalabile. Oltre al routing base, esistono funzionalità avanzate per ottimizzare caricamenti, sicurezza e gestione dei parametri.

6.1 Lazy Loading dei Moduli

- Permette di caricare i moduli solo quando necessario, migliorando le performance.
- Si configura nel modulo di routing principale con `loadChildren`.

```
// app-routing.module.ts
const routes: Routes = [
{ path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule) }
];
```

6.2 Route Guards

- Proteggono le rotte dall'accesso non autorizzato o gestiscono l'abbandono della pagina.
- Tipi principali:
  - `CanActivate`: controlla se una rotta può essere attivata.
  - `CanDeactivate`: controlla se si può lasciare una rotta.
  - `Resolve`: pre-carica dati prima di attivare la rotta.
  - `CanLoad`: controlla il caricamento di moduli lazy.

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

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
if (this.auth.isLoggedIn()) return true;
this.router.navigate(['/login']);
return false;
}
}
```

6.3 Parametri di Route e QueryParams

- **Parametri di route**: passati direttamente nell'URL, accessibili tramite `ActivatedRoute`.
- **QueryParams**: parametri opzionali nell'URL, ideali per filtri o paginazione.

```
// Parametro di route
this.route.params.subscribe(params => {
console.log(params['id']);
});

// Query params
this.route.queryParams.subscribe(q => {
console.log(q['filter']);
});
```

7 Gestione dello Stato Complesso in Angular

Gestire lo stato di un'applicazione complessa è fondamentale per mantenere dati sincronizzati tra componenti, ridurre bug e semplificare la manutenzione.

7.1 Strategie di Gestione dello Stato

- **Servizi Singleton**: memorizzano lo stato condiviso usando variabili interne o `BehaviorSubject`/`Subject`.
- **State Management Library**: librerie come NgRx, Akita o NGXS per gestire lo stato con pattern Redux.
- **LocalStorage / SessionStorage**: persistenza dei dati lato client.

7.2 Servizi con BehaviorSubject

- Utilizzabile per condividere dati tra componenti parent/child o componenti non correlati.
- Permette di osservare cambiamenti dello stato in tempo reale tramite Observable.

```
@Injectable({ providedIn: 'root' })
export class StateService {
private userSubject = new BehaviorSubject<User | null>(null);
user$ = this.userSubject.asObservable();

setUser(user: User) {
this.userSubject.next(user);
}
```

```
getUser(): User | null {
return this.userSubject.value;
}
}
```

7.3 Integrazione con componenti

- **Parent/Child**: il componente figlio si iscrive all'Observable per aggiornamenti automatici.
- **Componenti non correlati**: qualsiasi componente può iniettare il servizio e osservare i dati.

```
// child.component.ts
this.stateService.user$.subscribe(user => {
this.user = user;
});
```

7.4 NgRx / Redux pattern

- Basato su **Store**, **Actions**, **Reducers** e **Selectors**.
- Vantaggi: predicibilità, time-travel debugging, gestione complessa dello stato globale.

```
// Definizione action
export const loadUsers = createAction('[User] Load Users');

// Definizione reducer
export const userReducer = createReducer(
initialState,
on(loadUsersSuccess, (state, { users }) => ({ ...state, users }))
);

// Selettore
export const selectUsers = (state: AppState) => state.users;
```

7.5 Best Practices nella gestione dello stato

- Separare lo stato globale da quello locale del componente.
- Usare Observable e AsyncPipe per evitare subscribe manuali non necessari.
- Pulire sempre le subscription in `ngOnDestroy`.
- Scegliere librerie di gestione dello stato solo se l'app diventa complessa.
- Evitare modifiche dirette allo stato: usare sempre metodi del servizio o dispatch di action.

8 Animazioni in Angular

Angular offre un modulo dedicato `@angular/animations` per gestire animazioni fluide su componenti e elementi HTML, rendendo l'esperienza utente più dinamica.

8.1 Importare il modulo Animations

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
imports: [
BrowserAnimationsModule,
...
]
})
export class AppModule { }
```

8.2 Definizione di animazioni

- Usare il decoratore `@Component` con la proprietà `animations`.
- Importare funzioni da `@angular/animations`: `trigger`, `state`, `style`, `transition`, `animate`, `keyframes`.

```
import { trigger, state, style, transition, animate } from '@angular/animations';

@Component({
selector: 'app-box',
templateUrl: './box.component.html',
styleUrls: ['./box.component.css'],
animations: [
trigger('openClose', [
state('open', style({ height: '200px', opacity: 1 })),
state('closed', style({ height: '100px', opacity: 0.5 })),
transition('open <=> closed', [animate('0.5s ease-in-out')])
])
]
})
export class BoxComponent {
isOpen = true;

toggle() {
this.isOpen = !this.isOpen;
}
}
```

8.3 Usare le animazioni nel template

```
<div [@openClose]="isOpen ? 'open' : 'closed'" class="box">
  Contenuto animato
</div>
<button (click)="toggle()">Toggle</button>
```

8.4 Animazioni basate su trigger multipli

- È possibile definire più trigger nello stesso componente.
- Esempio: animazione di entrata e uscita con `*ngIf`.

```
trigger('fadeInOut', [
transition(':enter', [
style({ opacity: 0 }),
animate('300ms', style({ opacity: 1 }))
]),
transition(':leave', [
animate('300ms', style({ opacity: 0 }))
])
])
```

8.5 Keyframes e animazioni complesse

```
trigger('bounce', [
transition('* => *', [
animate('1s', keyframes([
style({ transform: 'translateY(0)', offset: 0 }),
style({ transform: 'translateY(-30px)', offset: 0.5 }),
style({ transform: 'translateY(0)', offset: 1.0 })
]))
])
])
```

8.6 Stati dinamici e parametri

- Possibile passare parametri dinamici alle animazioni.

```
trigger('openClose', [
state('open', style({ height: '{{height}}', opacity: '{{opacity}}' }), { params: { height: '200px', opacit
state('closed', style({ height: '100px', opacity: 0.5 })),
transition('open <=> closed', [animate('0.5s ease-in-out')])
])
```

8.7 Best Practices Animazioni Angular

- Preferire animazioni leggere per non degradare le performance.
- Usare `:enter` e `:leave` per animazioni di componenti dinamici.
- Riutilizzare trigger di animazione per uniformità UI.
- Evitare animazioni complesse in loop intensi o tabelle grandi.
- Testare le animazioni su diversi browser e dispositivi.

8.8 Stati dinamici e parametri

- Possibile passare parametri dinamici alle animazioni.
- Utile per altezza, opacità o durata che cambiano in base a dati runtime.

```
trigger('openClose', [
state('open', style({ height: '{{height}}', opacity: '{{opacity}}' }),
{ params: { height: '200px', opacity: 1 } }),
state('closed', style({ height: '100px', opacity: 0.5 })),
transition('open <=> closed', [animate('0.5s ease-in-out')])
])
```

8.9 Best Practices Animazioni Angular

- Preferire animazioni leggere per non degradare le performance.
- Usare `:enter` e `:leave` per animazioni di componenti dinamici.
- Riutilizzare trigger di animazione per uniformità UI.
- Evitare animazioni complesse in loop intensi o tabelle grandi.
- Testare le animazioni su diversi browser e dispositivi.
- Documentare trigger e parametri per facilitare manutenzione.

8.10 Stati dinamici e parametri

- Possibile passare parametri dinamici alle animazioni.
- Utile per altezza, opacità o durata che cambiano in base a dati runtime.

```
trigger('openClose', [
state('open', style({ height: '{{height}}', opacity: '{{opacity}}' }),
{ params: { height: '200px', opacity: 1 } }),
state('closed', style({ height: '100px', opacity: 0.5 })),
transition('open <=> closed', [animate('0.5s ease-in-out')])
])
```

8.11 Best Practices Animazioni Angular

- Preferire animazioni leggere per non degradare le performance.
- Usare `:enter` e `:leave` per animazioni di componenti dinamici.
- Riutilizzare trigger di animazione per uniformità UI.
- Evitare animazioni complesse in loop intensi o tabelle grandi.
- Testare le animazioni su diversi browser e dispositivi.
- Documentare trigger e parametri per facilitare manutenzione.

9 Gestione della Logica Asincrona con RxJS

Angular utilizza RxJS (Reactive Extensions for JavaScript) per gestire dati asincroni, flussi di eventi e comunicazione tra componenti in modo reattivo.

9.1 Concetti fondamentali di RxJS

- **Observable**: rappresenta un flusso di dati che può emettere valori nel tempo.
- **Observer**: oggetto che riceve i valori emessi da un Observable.
- **Subscription**: gestisce l'iscrizione a un Observable e permette di annullarla.
- **Operators**: funzioni per trasformare, filtrare o combinare flussi di dati (`map`, `filter`, `mergeMap`, `switchMap`, ecc.).

9.2 Creare un Observable

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

const obs = new Observable<number>(observer => {
observer.next(1);
observer.next(2);
observer.complete();
});

obs.subscribe({
next: value => console.log(value),
error: err => console.error(err),
complete: () => console.log('Completato')
});
```

9.3 Uso dei Subject

- `Subject` è sia un Observable che un Observer.
- Permette di multicasting dei valori a più subscribers.

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

const subject = new Subject<number>();
subject.subscribe(value => console.log('Subscriber 1:', value));
subject.subscribe(value => console.log('Subscriber 2:', value));

subject.next(1);
subject.next(2);
```

9.4 BehaviorSubject e State Management

- `BehaviorSubject` mantiene sempre l'ultimo valore emesso.
- Ideale per condividere stato tra componenti.

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

const state$ = new BehaviorSubject<number>(0);

state$.subscribe(value => console.log('Subscriber A:', value));
state$.next(1);
state$.subscribe(value => console.log('Subscriber B:', value)); // riceve subito 1
```

9.5 Operators comuni

- `map`: trasforma valori
- `filter`: filtra valori
- `tap`: esegue side-effect senza modificare il flusso
- `switchMap`: cancella flussi precedenti e ne sottoscrive uno nuovo (utile per chiamate HTTP)

- `mergeMap / concatMap`: unisce flussi senza cancellare quelli precedenti
- `catchError`: gestione errori

```
import { of } from 'rxjs';
import { map, filter } from 'rxjs/operators';

of(1, 2, 3, 4).pipe(
filter(x => x % 2 === 0),
map(x => x * 10)
).subscribe(console.log); // Output: 20, 40
```

9.6 Gestione chiamate HTTP con RxJS

```
this.http.get('/api/users').pipe(
map(users => users.filter(u => u.active)),
catchError(err => {
console.error(err);
return of([]); // valore di fallback
})
).subscribe(activeUsers => console.log(activeUsers));
```

9.7 Comunicazione tra componenti con RxJS

- Creare un servizio con `BehaviorSubject` per condividere dati tra componenti non parent-child.

```
// state.service.ts
@Injectable({ providedIn: 'root' })
export class StateService {
private user$ = new BehaviorSubject<User | null>(null);

setUser(user: User) { this.user$.next(user); }
getUser() { return this.user$.asObservable(); }
}

// componentA.ts
this.stateService.setUser({ name: 'Mario' });

// componentB.ts
this.stateService.getUser().subscribe(user => console.log(user));
```

9.8 Best Practices RxJS in Angular

- Usare `async pipe` nel template per sottoscrizioni automatiche.
- Pulire sempre le sottoscrizioni nei componenti con `takeUntil` o annullando `Subscription` in `ngOnDestroy`.
- Preferire operators come `switchMap` per evitare chiamate HTTP concorrenti inutili.
- Combinare flussi con `combineLatest`, `forkJoin` o `zip` quando necessario.
- Tenere la logica reattiva nel servizio e mantenere i componenti "leggeri".